RICE UNIVERSITY

Reasoning About Staged Programs

by

Jun Inoue

A Thesis Submitted
in Partial Fulfillment of the Requirements for the Degree

Master of Science

Approved, Thesis Committee:

________________________
Walid Taha, Chair
Assistant Professor of Computer Science
and Electrical & Computational
Engineering

________________________
Robert Cartwright
Professor of Computer Science

________________________
Vivek Sarkar
Professor of Computer Science and
E.D. Butcher Chair in Engineering

________________________
Edwin Westbrook
Post-Doctoral Fellow

Houston, Texas

May, 2010
ABSTRACT

Reasoning About Staged Programs

by

Jun Inoue

This thesis establishes formal equational properties of multi-stage calculi and related proof techniques that support analyses of staged programs. A key promise of staging is to make programs efficient without destroying clarity, thereby reducing the likelihood of bugs. However, few publications rigorously verify that their staged programs indeed behave as intended. In fact, little is known about how staged programs can be verified, or what correctness issues staging introduces. To solve this problem, I show a reduction of the correctness of a staged program to that of an unstaged program. This reduction not only clarifies the effects of staging on program behavior but also eases verification, as unstaged programs are more susceptible to existing reasoning techniques. I also demonstrate that important single-stage reasoning techniques apply to staged programs. These techniques are useful for establishing side conditions for the reduction and for discovering or validating further reasoning principles.
Acknowledgments

I would like to thank my advisor Walid Taha for introducing me to this thesis topic, for his support throughout the work, for the numerous opportunities he gave me to meet respected people in the field, and most of all for his enthusiasm for my work. He has always taken as much pride in my work as I have. I thank my thesis committee members, Robert Cartwright, Vivek Sarkar, and Edwin Westbrook. I thank Robert Cartwright for his advice on specific matters relating to this thesis. I thank Vivek Sarkar for his time and his support for this work. I thank Edwin Westbrook for his direct contributions to this work. Special thanks to Gregory Malecha, now at Harvard University, for the hints he has left me with. They proved to be essential for this work. I thank Mathias Ricken for his feedback that sharpened some of the results. I thank Eugenio Moggi, Yukiyoshi Kameyama, and Ronald Garcia for their valuable comments on the papers that I have written on precursors for this thesis. I thank Predrag Radosavljevic for sharing his \LaTeXthesis template with the Rice community, which I found useful for typesetting this thesis. This research was supported in part by NSF grant CCF-0747431.
Contents

Abstract ii
Acknowledgments iii
List of Figures vii
List of Tables viii

1 Introduction 1
  1.1 Why Staging ................................................. 1
  1.2 The Need for General-Purpose Reasoning Tools .............. 2
  1.3 Contributions and Organization ................................ 4

2 Multi-stage Programming 8
  2.1 Staging Annotations ........................................... 9
  2.2 Example: Gibonacci ........................................... 11
  2.3 Reasoning About Staged Programs Is Nontrivial ............ 12

3 The $\lambda^V$ Calculus 17
  3.1 Syntax and Semantics ......................................... 18
  3.2 Equational Theory ............................................ 26
    3.2.1 Provable Equality and Reduction ......................... 26
    3.2.2 Compatibility With Evaluation .......................... 31
  3.3 Anti-Equational Theory: What Are Not Equal .............. 34
    3.3.1 Generalized Reduction Axioms .......................... 34
    3.3.2 Substitution of Variables for Variables ............... 40
4 Effects of Staging on Correctness
   4.1 The Erasure Theorem .............................................. 44
   4.2 Example: Verifying Staged Gibonacci in Call-By-Name .......... 50
   4.3 Example: Verifying Staged Gibonacci in Call-by-Value ......... 53

5 Extensional Reasoning for \( \lambda^V \)
   5.1 Extensional Proof Principle ....................................... 61
   5.2 A Coinduction Primer ............................................. 68
   5.3 Deriving Applicative Bisimulation for \( \lambda^V \) ............... 71
      5.3.1 Howe’s Method ............................................... 71
      5.3.2 Adjusting to \( \lambda^V \) ...................................... 75
   5.4 Soundness and Completeness of Indexed Applicative Bisimulation .. 82
      5.4.1 Soundness ..................................................... 83
      5.4.2 Completeness ................................................ 87
      5.4.3 Correctness of Extensional Reasoning Principle ............. 90

6 Conclusion and Future Work ........................................... 92

Bibliography ............................................................... 95

A Proof Details ............................................................ 99
   A.1 Proof of \( e \Downarrow^\ell \iff [\Omega/x]e \Downarrow^\ell \) .................... 99
   A.2 Proof Details for Confluence ..................................... 102
   A.3 Proof Details of Standardization ................................. 106
   A.4 Proof Details for Anti-Equational Theory ....................... 115
   A.5 Proof Details for Example 5.11 ................................ 117
   A.6 Proof Details for Soundness and Completeness of Applicative
       Bisimulation ...................................................... 119
Figures

2.1 Staging Gibonacci. ................................................... 11

3.1 $\lambda^V$ syntax. ................................................... 18
3.2 Small-step semantics for $\lambda^V$. ................................. 21
3.3 Reduction and provable equality for $\lambda^V$. ..................... 27
3.4 Parallel reduction. ................................................... 30
3.5 Complete development. ............................................. 31
Tables

B.1 Summary of notations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126
Chapter 1

Introduction

This thesis establishes general reasoning principles for verifying the correctness of purely functional multi-stage programs. Whereas techniques for using staging effectively to improve performance are well-studied, little is known about how to check if the resulting programs are correct. I fill in this missing piece by proving equational properties of a multi-stage calculus, along with relevant proof techniques. These results ease rigorous verifications of the behaviors of staged programs and clarify the pitfalls of which multi-stage programmers should be aware.

1.1 Why Staging

Multi-stage programming (MSP) is a metaprogramming paradigm. A staged program executes in multiple stages, typically two. The earlier stage manipulates and generates code while the later stage executes the generated code. The extra code generation step provides an opportunity for adaptive and program-specific performance tuning that exploits the high-level structure of the program—something that automated optimizers do poorly. The performance benefits are often orders of magnitude greater than those that generic automated techniques can achieve [32].

The importance of these benefits is best summarized by the slogan “abstraction without guilt” [19]. Abstraction mechanisms like combinators, monads, classes, and first-class modules, which make a modern programming language “better” than its
predecessors, improve programmer productivity on the one hand but are a major source of runtime overhead on the other hand. MSP (and generative programming in general) resolves this dilemma: instead of directly running a high-level program, one can write a code generator that produces the same program with abstractions removed.

MSP has been effective in diverse applications. Verified combinatorial circuits [21], DSP code generation [22] and efficient programming language interpreters [5, 8] are prominent examples where high-level abstraction mechanisms ease or are expected to ease rigorous analysis of the programs’ correctness. In [22], the high-level clarity of the program even led to new insights on the algorithm. Along the same lines is high-performance computing [7], in which staging allows numerical computation to be highly parametrized without performance penalty. In all these areas, matching high-level clarity with efficiency is critical for building components which must be both reliable and efficient, and MSP helps the programmer achieve that goal.

1.2 The Need for General-Purpose Reasoning Tools

No matter how efficient high-level programs can be made with staging, all is lost if staging alters the behavior of the program and breaks its semantics. A metaprogramming tool (or any programming tool for that matter) is seriously deficient if it makes program validation difficult, as the whole point of high-level programming is to improve productivity by shielding programmers from bugs. In other words, trying to implement abstraction without guilt by such a tool would be self-defeating. An intuitive, formally specified semantics and some way of rigorously analyzing a metaprogram are therefore crucial.

Staging in the purely functional setting has a great advantage over other metapro-
gramming systems in this regard, namely the presence of a strong equational theory that delivers analysis capabilities on top of the performance enhancements. The \( \lambda^U \) calculus [31] shows that familiar reasoning principles from single-stage languages without code generation capabilities transfer to MSP. For example, \( \lambda^U \) justifies the identification

\[
(\lambda x.e) \ t = [t/x]e,
\]

where \( e, t \) are expressions with simple syntactic side conditions. The \( \lambda^U \) calculus permits such an equation to be applied not only in the generator code but also within the generated code. No other metaprogramming system seems to combine first-class code manipulation capabilities with equational reasoning of this strength. C++ templates [17], Cyclone [30], and DynJava [26] do not treat code as first-class values, while more sophisticated systems like LISP/Scheme macros [24, 9] have useless or unnatural equational theories [31].

The trouble is, however, that the equational theory is barely being exploited, and progress on reasoning techniques for MSP seems to have largely stopped with the invention of \( \lambda^U \). Apart from Pitts and Sheard’s attempt to reformulate Taha’s work in denotational style [27], which was apparently abandoned, no significant achievements seem to exist. Among the various MSP applications I cited in the previous section, only Brady and Hammond [5] even attempt to rigorously demonstrate that their staged program indeed behaves as specified. Although Brady and Hammond’s approach serves their purpose of constructing a verified domain-specific language implementation, it is unsatisfactory as a general-purpose reasoning mechanism. It relies on capturing the semantics of the program by dependent types, which may be extremely difficult to do in other settings.

Much of the theoretical expertise has so far been directed at developing safe type
systems for MSP, producing a dazzling variety of such systems [25, 33, 10, 20, 37, 18, 35, 36]. Generally, the goals of these systems are to guarantee well-typedness of generated code while maintaining expressivity and to understand the interplay between types and staging (and perhaps other features like side effects). Although these results definitely help to eliminate a large class of programming errors, in principle they can capture only a fragment of correctness for all staged programs and do not give insights on how to check a specific program’s correctness in its entirety.

I argue, therefore, that a general-purpose reasoning vehicle is needed to deliver abstraction without guilt, but that the reasoning vehicle is missing. The $\lambda^V$ calculus will make a fine foundation, but no specific advice on how to use this theory has ever been offered, and the properties of the equalities derivable in the theory are not well understood. This theory is like a bare programming language; the libraries and programming idioms that make it usable for specific tasks still need to be developed. The contributions of this thesis offer precisely those missing pieces, namely foundational theorems and proof techniques that help to verify specific staged programs.

### 1.3 Contributions and Organization

In this thesis, I prove metatheoretical properties of purely functional MSP languages that justify certain lines of equational reasoning. After informally reviewing the semantics of staging constructs in the MSP language MetaOCaml (Chapter 2), I present the $\lambda^V$ calculus (Chapter 3), which is a generalization of $\lambda^U$ [31]. The $\lambda^V$ calculus provides a set of equational axioms, one of which subsumes its counterpart in $\lambda^U$.

I formalize a purely functional core of MetaOCaml’s informal semantics as a deterministic, small-step semantics for $\lambda^V$ (Chapter 3). This semantics induces the central
subject of this study, the observational equivalence relation, which equates program fragments that are interchangeable under all contexts. I then present an axiomatic equational theory for $\lambda^V$ with the goal of proving a Plotkin-style [29] correspondence between the deterministic semantics and the equational theory:

- The equational theory equates a term and a value iff the term returns either that value or an equivalent value as output.

- The equational theory is a sound approximation of observational equivalence induced by the small-step semantics. In other words, the equational theory equates program fragments only if they are observationally equivalent.

The techniques used to prove these correspondences are identical to the ones in [31], and I have no claim to novelty in these proofs. Instead, my contribution is in discovering the very existence of a strengthened equational axiom that still fulfills the correspondence. Both the CBN and CBV semantics of $\lambda^V$ are studied.

I complement these results with a clear explanation of why some further generalizations to the equational axioms are not possible. Specifically, I formalize the concept of a level function, which describes how applying a context affects the level of a term. I prove that a rewrite rule that converts between contexts with incompatible level functions are never sound. I also discuss an important limitation that only arises in the CBV setting, namely that the equational theory cannot equate a function application $(\lambda x.e)\ y$, whose argument is a variable, with $[y/x]e$. I explain why this equation is unsound (and therefore underviable in the equational theory), and prove that a relaxed form of the equation is sound.

Using the equational theory just developed, I prove the Erasure Theorem, which establishes a direct correspondence between a provable equality of staged programs
and that of their unstaged counterparts (Chapter 4). Essentially, this theorem states that syntactically erasing staging constructs is a map from CBN $\lambda V$ to CBN $\lambda V$ that preserves provable equality, i.e. an endomorphism on CBN $\lambda V$. Erasure of staging constructs on CBV $\lambda V$ terms is also a homomorphism, but the codomain is CBN $\lambda V$, not CBV. As CBN provable equality is a superset of CBV provable equality, the CBV statement is weaker and less useful than in CBN. A refined version of the theorem shows a restricted notion of provable CBV equality that erasure maps to ordinary CBV equality.

The Erasure Theorem (or Theorems) reduces the correctness of a staged program to that of an unstaged program. This reduction is important because an unstaged program is susceptible to reasoning techniques developed for single-stage calculi and is therefore much easier to analyze. The preconditions for applying these theorems give an exhaustive characterization of the bugs that staging annotations can introduce. Hence I show both the “do”s and “do not”s of reasoning about the effects of staging on program behavior. The Erasure Theorem technique applies most straightforwardly to a partial-evaluation style of staging, where the programmer starts with an unstaged program and adds staging annotations to it with minimal intrusion to the overall program logic. The technique should also apply to other contexts with suitable arrangements.

Probing further for equational properties of multi-stage languages, I prove that a coinductive re-characterization of observational equivalence called applicative bisimulation [1, 15, 13], which had previously only been studied in single-stage calculi, exists in $\lambda V$ as well (Chapter 5). Applicative bisimulation induces a coinductive equational reasoning principle that is both sound and complete, that is, it proves precisely the set of all valid observational equivalence assertions. The coinductive principle is harder
to use than the axiomatic equational theory and is also weaker than its counterpart in a single-stage language but still validates some useful equivalences that are otherwise difficult to prove. In fact, the axiom that $\lambda^V$ generalizes over $\lambda^U$ was first justified with this principle.

I use the coinductive reasoning principle to prove extensionality, which states that functions behaving identically on all inputs must be observationally equivalent. This result is intuitively obvious, but in most calculi, including $\lambda^V$, the proof is nontrivial. Extensionality revives one of the generalizations once rejected in Chapter 3, namely substitution of variables into function bodies, under mildly restricted contexts. This success concretely demonstrates the potential of the coinductive theory to extend equational reasoning beyond the capabilities of the axiomatic equational theory.

Although the theory developed in this thesis is fairly comprehensive, there is still much room for future work. I conclude this document with a review of what has and has not been achieved in this thesis along with a discussion of potential directions for future research (Chapter 6).

Due to the nature of the work, this thesis contains many rigorous proofs. In the interest of conciseness, I have suppressed the details of these proofs in the main text, except where they help to illustrate a point. The missing details for non-trivial proofs are collected in Appendix A. Mathematical notations are summarized in Appendix B for the reader’s convenience.

Taken together, these chapters constitute a coherent body of work that supports my thesis: with a degree of care to a specific set of technical details, equational reasoning for multi-stage programs is feasible or even easy. Therefore, multi-stage programming can indeed help programmers improve the performance of their programs while preserving clarity and correctness.
Chapter 2

Multi-stage Programming

This chapter informally introduces multi-stage programming using the MSP language MetaOCaml [6]. This language extends the functional language OCaml with three annotation constructs, brackets, escape, and run, which direct how to split up a program’s execution into multiple stages. I present a concrete example, the Gibonacci function, which I will demonstrate how to verify in Chapter 4. Although other chapters of the thesis, including Chapter 4, almost exclusively work with the core λV calculus instead of MetaOCaml, I introduce the programming examples with MetaOCaml to emphasize that my results pertaining to the core calculus tie in directly with an actual programming language implementation. Furthermore, MetaOCaml’s syntax offers superior readability for large expressions such as complete programs and is therefore better suited for presenting concrete examples.

After illustrating the informal semantics with MetaOCaml, I will validate my work by showing a series of examples that illustrate some of the important semantic subtleties that staging can introduce. I complement each of these examples with a clear characterization of the extent to which the subtleties can affect correctness and the conditions which guarantee that correctness is unaffected.
2.1 Staging Annotations

MetaOCaml is an extension of OCaml with three language constructs: brackets (.<>.), escape (.~), and run (.!). MetaOCaml is impurely functional, but the focus of this thesis is the purely functional subset.

Brackets generate code. For example,

```
.<(fun x -> x * x) 2>.
```

evaluates to a code value, i.e. the parse tree of the expression `((fun x -> x*x) 2)`, and not to the integer value 4. The type of this code value is `<int>`, read “code of int”. Just like quasiquotation in LISP-like languages, `<>`, gives a syntax for writing down code as a literal value.

Escapes allow parts of code values to be computed dynamically, similarly to LISP unquote. For example:

```
let double x = .<.~x * 2>. in
double .<1 + 3>.
```

When the second line calls the `double` function, `.<1+3>`. gets substituted for `x` to produce the intermediate term `.<.~(.<1+3>.) * 2>`. Then `.~` merges the code value `.<1+3>`. into the surrounding, which results in the code value `.<(1+3) * 2>`. being returned. The `x` inside `.~` can be any other expression that returns a code value of the right type. An `.~` can only appear (lexically) inside `<>`. The `<>`. and `.~` can be nested, but `.~` cannot be nested deeper than `<>`. The depth of `<>`. minus the depth of `.~` is called the level. Execution of escaped code is deferred like its surrounding if the nesting of `<>`. is strictly deeper (i.e. level > 0).

*Actually, MetaOCaml assigns it the type (`'a, int`) code where `'a` is an environment classifier [33]. Classifiers are orthogonal to the results presented here and are therefore omitted in all types.
Finally, we have an analogue of LISP’s eval function, which is run (.!). This construct takes a code value, compiles it into machine code (or byte code, depending upon the flavor of MetaOCaml being used), and immediately executes the machine (or byte) code. For example,

\[
.!(\langle\text{fun } x \rightarrow x \times x\rangle\ 2).
\]

returns 4.

Resolving escapes and generating a code value is called stage 0, while executing the generated code with .! is called stage 1. Stage 1 code may also create and run another code value, which constitutes stage 2, but stages 2 and later are rarely useful.

Staging supports cross-stage persistence (CSP), which allows a value created at one stage to be used in any subsequent stage. For example, the following is a valid MetaOCaml program.

\[
\text{let } \text{two} = 2 \text{ in} \\
\text{let } \text{double} \ x = x \times \text{two} \text{ in} \\
.\langle\text{double } 5\rangle.
\]

This program returns the code value \(\langle\text{fun } x \rightarrow x \times 2\rangle\ 5\rangle\), which captures a user-defined value double. In the formal calculus, this capturing is expressed by substitution, as I just showed. In the implementation, the code value is represented as \(\langle\square \ 5\rangle\), where \(\square\) is a pointer to a closure object.

MSP has some important features that distinguish it from most other metaprogramming systems. Unlike its LISP counterparts, staging annotations guarantee that the generated code is well-formed, whereas a LISP macro can generate nonsensical code fragments such as \((\text{lambda } 1)\). Staging also offers automatic hygiene like Scheme macros [9] but unlike traditional LISP macros. Staging also guarantees type
safety for purely functional programs, including safety of the generated code [33].
Recently, several works have expanded type safety to languages with imperative fea-
tures [20, 18, 36]. With these strong theoretical foundations staging is particularly
amenable to analysis, which makes it an attractive system in which to program.

2.2 Example: Gibonacci

(* Unstaged *)
(* gib : int → int → int → int *)
let rec gib n x y =
   if n = 0 then x else
   if n = 1 then y else
   let r1 = gib (n-1) x y
   and r2 = gib (n-2) x y in
   r1 + r2
let gib_4 = gib 4

(* Staged *)
(* gibgen : int → <int> → <int> → <int> *)
let rec gibgen n x y =
   if n = 0 then x else
   if n = 1 then y else
   .<(~(gibgen (n-1) x y) + ~(gibgen (n-2) x y))>.

(* gibst : int → int → int → int *)
let gibst n =
   .!.<fun x y → ~(gibgen n .<x>. .<y>.)>.
let gibst_4 = gibst 4

(* execution examples *)
gibst 0 2 3 ⇒ 2
gibst 4 2 3 ⇒ 8
(* assuming x,y are bound: *)
gibgen 0 .<x>. .<y>. ⇒ .<x>.
gibgen 4 .<x>. .<y>. ⇒ .<(((y+x)+y)+(y+x))>.

Figure 2.1 : Staging Gibonacci.
Figure 2.1 shows staging in action. The \texttt{gib} function is an (unstaged) implementation of the Generalized Fibonacci (Gibbonacci) sequence. Gibonacci is a sequence defined by the same recurrence as the Fibonacci sequence but the beginning of the sequence is parametrized as \texttt{“x, y, x+y, ...”} instead of being fixed as \texttt{“0, 1, 1, ...”}. The \texttt{gib} function is just a transcription of the mathematical definition into OCaml code. Adding staging annotations to \texttt{gib} gives the code generator \texttt{gibgen}. An invocation like
\begin{verbatim}
gibgen 4 .<x> .<y>.
\end{verbatim}
produces a code value representing the computation that \texttt{gib 4 x y} performs, except with all the recursion unrolled and branching eliminated.

The \texttt{gibst} function is a wrapper to \texttt{gibgen} that takes just the index \texttt{n} of the sequence element to compute and creates a function that maps \texttt{x} and \texttt{y} to the \texttt{n}-th element of the Gibonacci sequence for that \texttt{x-y} pair. In the example, we use this wrapper to create a function \texttt{gibst_4} that is specialized for \texttt{n=4}. The main difference between \texttt{gib_4} and \texttt{gibst_4} is performance. \texttt{gib_4} is simply less general than \texttt{gib}, and barring clever compiler optimizations, it is no faster than calling \texttt{gib} directly. By contrast, \texttt{gibst_4} incurs a code generation and compilation cost when it gets bound, but after that it runs as efficiently as if the programmer had manually written the recursion-unrolled code.

### 2.3 Reasoning About Staged Programs Is Nontrivial

As explained in Section 2.1, a major advantage of MSP over other programming systems is the armory of strong guarantees that shield the programmer from mistakes. Despite all of those properties that make MSP particularly wieldy to the programmer,
MSP semantics is surprisingly subtle to reason about, and untrained intuition alone can be insufficient to guide reasoning over multi-stage code.

For example, a naïve expectation that a novice multi-stage programmer—such as myself, as of a few years ago—might have is that since staging annotations only control when things are done and not what is done, they cannot alter the result of executing a purely functional program. But consider the code below.

```ocaml
let rec loop () = loop () (* infinite loop *)
let f x = (let _ = loop () in 0)
let g x = (let _ =..<loop ()>. in 0)
```

The functions \(f\) and \(g\) differ only by staging annotations in their bodies, but \(f\) is nowhere defined whereas \(g\) is everywhere defined as 0. Thus, staging annotations can make a program more terminating. To complement this observation, consider the following code.

```ocaml
let h b = if b then (loop ()) else 0
let i b = .!..<if .~b then .~(loop ()) else 0>. 
```

Now \(h\) is defined on the argument \(false\) while \(i\) is nowhere defined. Thus staging can make a program less terminating as well.

Note that the disparity between \(f\) and \(g\) in the first example relies on MetaOCaml’s evaluation strategy being call-by-value (CBV): in call-by-name (CBN), both \(f\) and \(g\) terminate and return 0 on all inputs. Is it possible, then, for staging annotations to make a CBN program more terminating? The following example answers this question in the affirmative:

```ocaml
let f x = loop ()
let g x =..<loop ()>. 
```
However, this example misses the point. Allowing $g$ to return a code value gives $g$ an unfair advantage, because any computation can be suspended by just wrapping it in brackets. The function $g$ recovers termination by simply refusing to do anything interesting. How about a more refined question, such as: can staging annotations make a program more terminating if the return value of the annotated program is required to be a staging-unrelated value, like an integer? The answer to this question is not obvious by intuition (although perhaps intuition can make a good guess).

One of the key results of this thesis, the CBN Erasure Theorem, gives a definitive negative answer to this question (Chapter 4). If an annotated CBN program terminates to a staging-unrelated value, then the unannotated, or unstaged, counterpart must terminate to the same value.

Another interesting thing to observe about these examples is that only the termination behavior is changed by staging, which leads us to another question: is it ever possible for staging to alter the semantics of a program without affecting termination? For example, can a program that returns 1 be made to return 2 by adding staging annotations? The answer to this question again lies in the Erasure Theorem, which guarantees that staging annotations in the program cannot alter the return value, except perhaps for left-over staging annotations. For example, a program that returns 1 cannot be annotated into a program that returns 2, although it can perhaps be annotated into one that returns $<4 - 3>$.

General equations are even more subtle. For example, suppose an expression $! .<e> .$, where $e$ is some expression, appears as a part of a program. A run cancels a pair of brackets, so is it safe to optimize this fragment as $e$ before program execution? In other words, does replacing $! .<e> .$ by $e$ always preserve the outcome of the overall program? The answer turns out to be negative. See the following example.
The function f terminates because the nesting of escape is strictly shallower than brackets around loop (); so the loop () is not executed. By contrast, when g is called, it immediately starts executing the loop () and fails to terminate. So this conversion is not always safe.

Consider a similar example, .~.<e>. It turns out that this expression is always safe to replace by e. No pathological example like the above exists, where a program using .~.<e>.. terminates but the same program using e instead fails to terminate, or vice versa. This result is nontrivial; it was previously only known to be safe if e belongs to a class of expressions called level-0 expressions (see Chapter 3). One of the concrete contributions of this thesis is a proof (actually, two proofs) that this conversion is safe for any expression e.

Consider yet another similar example, .!<.~e>. Is it safe to rewrite this expression as e? The answer is no, at least in the untyped setting. But the answer for a slight modification is yes: rewriting .!.<.~e>. as .!e is always safe. These issues are discussed in depth in Section 3.3.

As these examples show, staging has its own share of subtle correctness issues that, at a glance, may seem quite tricky. However, as the conciseness and clarity of my answers to these questions may indicate, the equational reasoning that I develop in this thesis makes these subtleties considerably easier to handle.

In fact, I attribute the difficulty in trying to answer these questions by intuition not to inherent complexity in MSP but to the lack of in-depth understanding of this aspect of MSP. Older, more widely used metaprogramming systems such as C++ templates or LISP/Scheme macros have their own setbacks, but those problems do
not prevent programmers from effectively using these tools because the important pitfalls are well-known and documented. Similarly, as the understanding of MSP semantics deepens and as knowledge about the important issues accumulates, casual reasoning (in addition to formal reasoning) of MSP should become even easier. This thesis takes a significant step forward in this direction by giving the first sizable batch of results, including tools for further general theoretical investigation.
Chapter 3

The $\lambda^V$ Calculus

As with any practical programming language, MetaOCaml’s full feature set is too large to capture and analyze mathematically. In this chapter, I present the multi-stage $\lambda$ calculus, $\lambda^V$, which formalizes a purely functional core of MetaOCaml. This calculus is a strengthened version of Taha’s $\lambda^U$ [31], and is obtained by generalizing the equational axiom for resolving escapes. I present the deterministic semantics and axiomatic equational theory. The purpose of this chapter is to prove Plotkin-style correspondence between the deterministic semantics and the equational theory in both evaluation strategies. More specifically, I prove that the axiomatic semantics is a sound approximation of observational equivalence that is strong enough to trace program execution.

I prove these results for both the CBV ($\lambda^V_v$) and CBN ($\lambda^V_n$) variants. Most of the definitions and proofs are independent of evaluation strategy, with the differences between CBV and CBN factored out into a few key definitions. The following notation will be used to specify the evaluation strategy where necessary.

**Notation.** A set (possibly a relation) $S$ may be marked as CBV ($S_v$) or CBN ($S_n$) if its definition varies by evaluation strategy. The subscript is dropped in assertions and definitions that apply to both evaluation strategies.
Levels
\[ \ell, m \in \mathbb{N} \]

Variables
\[ x \in \text{Var} \]

Expressions
\[ e \in E ::= x \mid \lambda x.e \mid e \ e \mid \langle e \rangle \mid \neg e \mid \text{run } e \]

Stratified Expressions
\[ e^0 \in E^0 ::= x \mid \lambda x.e^0 \mid e^0 e^0 \mid \langle e^1 \rangle \mid \text{run } e^0 \]
\[ e^{\ell+1} \in E^{\ell+1} ::= x \mid \lambda x.e^{\ell+1} \mid e^{\ell+1} e^{\ell+1} \mid \langle e^{\ell+2} \rangle \mid e^\ell \mid \text{run } e^{\ell+1} \]

Programs
\[ p \in \text{Prog} ::= e^0 \text{ [closed]} \]

Values
\[ v^0 \in V^0 ::= \lambda x.e^0 \mid \langle e^0 \rangle \]
\[ v^{\ell+1} \in V^{\ell+1} ::= e^\ell \]

Contexts
\[ C \in \text{Ctx} ::= \bullet \mid \lambda x.C \mid C \ e \mid e \ C \mid \langle C \rangle \mid \neg C \mid \text{run } C \]

Figure 3.1 : \( \lambda^V \) syntax.

3.1 Syntax and Semantics

Figure 3.1 shows the syntax of \( \lambda^V \), which is identical to \( \lambda^U \). It extends the plain \( \lambda \) calculus with the three staging constructs. Expressions are stratified into levels. A level is a natural number, denoted \( \ell \) or \( m \), and the set of all expressions at level \( \ell \) is
written $E^\ell$. An expression is at level $\ell$ iff the nesting of escapes is no more than $\ell$ levels deeper than brackets at any point in the expression. In other words, a term’s level is the depth of nesting of escapes, with the rule that a bracket and an escape cancel with each other if the bracket encloses the escape. Examples are $\bar{x} \in E^1$, $\langle \bar{x} \rangle \in E^0$, $\bar{\langle x \rangle} \in E^2$, and $\bar{\langle \langle x \rangle \rangle} \in E^1$.

**Remark 3.1.** Note that $E^0 \subseteq E^1 \subseteq \cdots \subseteq E^\ell \subseteq \cdots \subseteq E$, and that the $E^\ell$ are subsets of $E$ and not distinct from $E$. The superscript on $e^\ell$ is a constraint on the shape of the expressions that $e^\ell$ ranges over and not an annotation that forms a part of the term.

A program is a closed level-0 expression, i.e. an expression in which all variables are bound and escapes are nowhere nested deeper than brackets. Thus an expression is at level $\ell$ iff it is allowed to appear within $\ell$ brackets in a legal program. The evaluation of a program is also stratified into levels. An evaluation step happens at level $\ell$ iff it takes place within $\ell$ brackets. Likewise, an expression $e$ occurs at level $\ell$ iff it occurs within $\ell$ brackets in a legal program, which in turn is permitted iff $e$ is at level $\ell$ in the sense of being a member of $E^\ell$.

Values at a given level $\ell$ are expressions that contain no more work at that level, i.e. expressions that are considered fully evaluated if they appear within $\ell$ brackets. Naturally, values at level $\ell$ are expressions at level $\ell$ with additional constraints. For level 0, “work” means function application, code generation, and code execution by `run`, where work within the body of an abstraction is delayed. A level-0 value is therefore either an abstraction or a code with no escapes to be resolved. Note that an abstraction must have a level-0 body since a level-1 expression that is not also level-0 would fail to make the whole abstraction be level-0. For level 1, work consists of executing escaped level 0 work and splicing the result into the surroundings. For
higher levels, work consists of looking for escapes and executing the lower-level work contained therein. In either case, for all levels $\ell > 0$, the values are expressions at level $\ell - 1$, which are just the expressions that do not nest escapes $\ell$ times.

A context is an incomplete expression $C$ with exactly one hole $\bullet$. I write $C[e]$ to denote the complete expression obtained by replacing the hole of $C$ with $e$.

**Definition 3.2** (Term And Context Size). Term size is written $\text{size}(e)$. Define the size$(C)$ of a context as $\text{size}(C) \overset{\text{def}}{=} \text{size}(C[x]) - 1$. Note the hole counts as size 0 while the sizes of subexpressions of $C$ are included, e.g. $\text{size}(\bullet \langle x \rangle)$ is 3, not 1.

**Notation.** I use $(=)$ for $\alpha$ equivalence. Let $\text{FV}(e)$ be the set of all free variables in $e$. Metavariables $t$ and $d$ (in addition to $e$) range over expressions unless otherwise specified. Any superscripts on them indicate constraints on their level. Similarly, $u$ and $w$ stand for values.

Note that syntactic equality need not be indexed by $\cdot_n$ or $\cdot_v$ because the syntax is identical for both CBN and CBV. In particular, $V^0$ does not contain $\text{Var}$ in CBV. I will come back to this point in Section 3.3.

Figure 3.2 shows the small-step semantics for $\lambda^V$. Primitive small-steps $(\overset{\text{prim}}{\sim}_m)$ specify how each form of redex reduces. SS-$\beta$ performs standard $\beta$ reduction at level 0, where the argument must be a substitutable expression, i.e. a $t^0$ for CBN, or a $v^0$ for CBV. SS-$E_V$ resolves escapes at level 1, and SS-$R_V$ executes a run at level 0. A small-step $(\overset{\sim}_{\ell})$ at level $\ell$ consists of a primitive small-step at level $m$ inside an $\ell, m$-evaluation context $E_{\ell,m}$, which takes a level-$m$ expression (the redex or its reduct) and yields an expression at level $\ell$. The evaluation context picks out an outermost, leftmost redex. A CBV evaluation context $E_v$ places a hole at or in the function argument at all levels, but CBN does not evaluate function arguments at level 0.
Evaluation Contexts

\[ E^{0,m} \in ECtx^{0,m} ::= \bullet[m = 0] \mid E^{0,m} \cdot e^0 \mid \langle E^{1,m} \rangle \mid \text{run } E^{0,m} \]
\[ \mid v^0 \cdot E^{0,m}[\text{only in CBV}] \]
\[ E^{\ell+1,m} \in ECtx^{\ell+1,m} ::= \bullet[\ell + 1 = m] \mid \lambda x. E^{\ell+1,m} \cdot E^{\ell+1,m} \cdot e^{\ell+1} \mid v^{\ell+1} \cdot E^{\ell+1,m} \]
\[ \mid \langle E^{\ell+2,m} \rangle \mid \overline{E}^{\ell,m} \mid \text{run } E^{\ell+1,m} \]

Production rules marked as \([\phi]\) apply only if \(\phi\) holds.

Substitutable Arguments

\[ a, b, c \in A \overset{\text{def}}{=} \begin{cases} E^0 & (\text{in CBN}) \\ V^0 & (\text{in CBV}) \end{cases} \]

Redexes

\[ r^0 ::= (\lambda x.e^0) \cdot a \mid \text{run } \langle e^0 \rangle \]
\[ r^1 ::= \overline{\langle e^0 \rangle} \]

Small-step Rules

\[ (\lambda x.e^0) \cdot a \overset{\text{prim}_0}{\sim} [a/x]e^0 \]
\[ \overline{\langle e^0 \rangle} \overset{\text{prim}_1}{\sim} e^0 \]
\[ \text{run } \langle e^0 \rangle \overset{\text{prim}_0}{\sim} e^0 \]

\[ e \overset{\text{prim}_m}{\sim} t \]
\[ E^{\ell,m}[c] \overset{\ell}{\sim} E^{\ell,m}[t] \]

Figure 3.2 : Small-step semantics for \(\lambda^V\).

At higher levels, CBN also evaluates function arguments because they may contain escapes.

Notation. \((\overset{\text{prim}}{\sim}) \overset{\text{def}}{=} (\overset{\text{prim}}{\sim}_0) \cup (\overset{\text{prim}}{\sim}_1)\). For a binary relation \(R\), let \(R^*\) and \(R^+\) be the
reflexive-transitive and transitive closures of $R$, respectively. I write $x R^{n+1} y$ iff there is a sequence $x R z_1 R z_2 R \cdots R z_n R y$ and let $R^0$ be equality.

This semantics satisfies the usual properties expected of a small-step semantics. Proposition 3.3 states that a value does not step, so it does not embody any work. Proposition 3.4 proves that small-step is deterministic. Proposition 3.5 states that the hole of an evaluation context specifies where the next small-step (if any) must happen.

**Proposition 3.3.** $\ell^\ell \neq \mathcal{E}^{\ell,m}[t^m]$.

*Proof.* Straightforward induction on $\mathcal{E}^{\ell,m}$. \hfill \Box

**Proposition 3.4.** $(e \sim^\ell t_1 \land e \sim^\ell t_2) \implies t_1 = t_2$.

*Proof.* Straightforward induction on the derivation of $e \sim^\ell t_1$. \hfill \Box

**Proposition 3.5.** An expression $\mathcal{E}^{\ell,m}[e^m]$ where $e^m \not\in V^m$ small-steps at level $\ell$ iff $e^m$ small-steps at level $m$.

*Proof.* The “if” direction follows from the fact that composing any pair of evaluation contexts yields another evaluation context. Formally, this direction is proved by trivial induction on $\mathcal{E}^{\ell,m}$. For the “only if” direction, $\mathcal{E}^{\ell,m}[e^m] = \mathcal{E}'[r^n]$ for some $\mathcal{E}' \in ECtx^{\ell,n}$. Then straightforward induction on $\mathcal{E}^{\ell,m}$ shows that there is some $\mathcal{E}'' \in ECtx^{m,n}$ such that $\mathcal{E}^{\ell,m}[\mathcal{E}''[n]] = \mathcal{E}'$ and, therefore, $\mathcal{E}''[r^n] = e^m$. \hfill \Box

These propositions have the important consequence that for an expression $\mathcal{E}[e]$ to terminate, $e$ must terminate first. This lemma is used in several parts of the proof to show that certain terms are forced to diverge.

**Lemma 3.6.** If $e^\ell = \mathcal{E}^{\ell,m}[t^m] \sim^\ell n v \in V^\ell$, then $t^m \sim^m n' u \in V^m$ where $n' \leq n$. 

Proof. By Proposition 3.5, $e^\ell$ keeps small-stepping to expressions of the form $E^{\ell,m}[d^m]$, with only $d^m$ changing at each step, until $d^m$ becomes a value. By Proposition 3.4, these steps must form a prefix of $e^\ell \leadsto^\ell n v$, so $n' \leq n$.

The central notion of equivalence for program fragments is observational equivalence. This relation captures the intuitive idea of substitutability of equals, namely that replacing a (sub)expression with an equivalent should not alter the observable behavior of the whole program. Since my treatment of $\lambda V$ omits constants, observation is restricted to termination at level 0.

**Definition 3.7** (Termination and Divergence). Say that a term $e \in E^\ell$ *terminates* at level $\ell$ to the value $v \in V^\ell$, written $e \Downarrow^0 v$ (or just $e \Downarrow^\ell$ if $v$ is unimportant), iff $e \leadsto^\ell v$. If no such $v$ exists, say that $e$ *diverges* (at level $\ell$), written $e \Uparrow^\ell$. If $e$ small-steps infinitely many times, then $e$ is said to be *non-terminating*. An $e$ is *stuck* at level $\ell$ iff it neither small-steps nor is a value at that level. An $e$ gets stuck iff it small-steps to a stuck term.

**Notation.** Let the metavariable $\Omega$ range over divergent terms, in particular those that diverge at level 0 unless otherwise specified.

**Definition 3.8** (Observational Equivalence).

$$e \cong t \overset{\text{def}}{=} \forall C. (C[e], C[t] \in \text{Prog} \implies (C[e] \Downarrow^0 \iff C[t] \Downarrow^0))$$

In this definition, an observation consists of placing a term within a program context and testing whether the resulting program terminates, and two terms are equivalent iff no observation distinguishes them. A context $C$ that proves a pair of expressions to be nonequivalent is called a distinguishing context for that pair. Although by definition only closing contexts are considered interesting, it is important to note that staging forces us to consider non-closing contexts as well.
**Notation.** A sequence, or family, of mathematical objects $x_i$ indexed by a set $I$ is written $\tau_{i \in I}$. I write $\lambda x_{i \in \{1, \ldots, n\}}$ for a sequence of nested binders $\lambda x_1. \lambda x_2. \ldots \lambda x_n$. The superscript binds the index variable $i$. The superscript may be abbreviated like $\tau_i$ or omitted if the intention is clear. Let the variable name $\_$ be reserved for dummy bindings, i.e. in $\lambda \_. e$, the variable $\_$ does not appear free in $e$.

**Definition 3.9.** Define an open observational equivalence (\(\approx_{\text{op}}\)) just like (\(\approx\)) but using $E^0$ in place of $\text{Prog}$. Formally,

\[
e \approx_{\text{op}} t \iff \forall C. \ (C[e], C[t] \in E^0 \implies (C[e] \Downarrow^0 \iff C[t] \Downarrow^0)).
\]

**Proposition 3.10.** \((\approx) = (\approx_{\text{op}})\).

**Proof.** I will prove the CBV case first. Clearly $e \approx_{\text{op}} t \implies e \approx_\nu t$. For the converse, suppose $e \not\approx_{\text{op}} t$, and let $C$ be a (not necessarily closing) context that distinguishes $e$ and $t$. Let $\lambda \tau_i$ denote a sequence of $\lambda$’s that bind all free variables in $e$ and $t$. Let $d_1;d_2$ denote sequencing, which checks that $d_1$ terminates, discarding the return value, and then evaluates $d_2$. Sequencing is just syntactic sugar for $(\lambda \_. d_2)$ $d_1$ in CBV. Then the context $C_0 \overset{\text{def}}{=} (\lambda \tau_i. \langle \lambda \tau_i; \langle C; (\lambda y.y) \rangle \rangle)$ is a closing context that distinguishes $e$ and $t$, so $e \not\approx_\nu t$.

Now consider CBN. Again, obviously $e \approx_{\text{op}} n t \implies e \approx_n t$. Suppose for the converse that $e \not\approx_{\text{op}} n t$, and let $C$ be a distinguishing context. The justification for \((\approx_n) = (\approx_{\text{op}})\) rests on the fact that a term $e$ terminates iff $[\Omega/x]e$ does for a closed, level-0, divergent $\Omega$. A detailed proof of this fact is found in Section A.1. Therefore $C_1 \overset{\text{def}}{=} (\lambda \tau_i; C) \Omega_i$ distinguishes $e$ and $t$, where $\Omega$ is applied as many times as there are $x_i$.

The first half of this proof shows that $\lambda \nu$ effectively forces observation of open terms because evaluation can go under binders at level $> 0$. Hence reasoning about
the equivalence of expressions must directly handle open terms, and the standard trick of substituting away free variables does not work because the free variables can be genuinely free during program execution. In this sense, \( \lambda^V \) has an inherently open-term semantics. A further complication in the untyped \( \lambda^V \) is that these variables are not necessarily quoted by brackets if the bindings are ill-leveled, as in the context \( C_0 \) for the CBV proof above. A variable is ill-leveled iff it is bound at level \( \ell \) and used at some level \(< \ell \). An ill-leveled variable reference is considered a perverse usage of staging because it is like asking for the value of a variable before it is supplied.

The latter half of the proof shows that open and closed observations coincide because a closing context can simulate the presence of free variables by injecting divergence instead, as \( C_1 \) does. This argument does not rely on staging features and the same conclusion holds for the plain \( \lambda \) calculus. However, this proof does not show that \( \lambda^V_n \) is exempt from the complication of reasoning directly about open terms. The trick used for CBV works in CBN to some extent, although not completely because sequencing is not available: in CBN, evaluation of \( C[e] \) and \( C[t] \) must be forced explicitly by somehow using their return values. Without loss of generality, suppose \( C[e] \Downarrow^0 \) and \( C[t] \Uparrow^0 \). If \( C[e] \Downarrow^0 \lambda x.d^0 \), then \( C_2 \overset{\text{def}}{=} \langle \lambda x.\overline{C} \rangle \) distinguishes \( e \) and \( t \). If \( C[e] \Downarrow^0 \lambda x.d^0 \), then \( C_3 \overset{\text{def}}{=} \langle \lambda x.\overline{C} \rangle \) almost distinguishes \( C[e] \) and \( C[t] \) but cannot tell apart \( \lambda x.\Omega \) and \( \Omega \).

As \( C_2 \) and \( C_3 \) demonstrate, CBN execution of a staged program can involve open-term execution just like in CBV. The open-term semantics is forced by the ability for an evaluation context to place the hole under a binder, and \( \lambda^V_n \) also has this property. All that \( C_1 \) proves is that the distinguishing power of free variables is not as strong as in CBV, and that for the purpose of measuring the size of \( (\approx_n) \) the effects of free variables is masked by the effects of substitution of divergent terms.
Proposition 3.10 establishes that \((\approx_{\text{op}})\) is a valid alternative definition for \((\approx)\) in \(\lambda^V\). For the rest of this document I take the open-term version, Definition 3.9, to be the definition of \((\approx)\), since the additional closedness check in Definition 3.8 is pointless in an open-term semantics. In the rest of this chapter, I wish to demonstrate with the inductive equational theory that the open term semantics nonetheless interferes little with equational reasoning. The one problem in this development is the lack of the ability to treat variables as substitutables, but this issue is resolved to a large extent by the extensional theory to be presented in Chapter 5.

3.2 Equational Theory

An equational theory (also called axiomatic semantics) is a basic tool for metatheoretical investigation, particularly establishing program equivalences. The \(\lambda^V\) calculus comes with a strong equational theory inherited from \(\lambda^U\), which is a major advantage over other program generation systems like LISP macros whose equational theory is trivial [31]. In this section I present the proof rules for the equational theory and prove that the theory induces a confluent rewrite system. I show that the set of equations provable in this theory is a sound approximation of observational equivalence.

The theory of \(\lambda^V\) is mostly identical to that of \(\lambda^U\), with one key difference. In \(\lambda^U\), a term \(\tau\langle e \rangle\) can be equated to \(e\) only if \(e \in E^0\), but \(\lambda^V\) removes this restriction and allows this equation for all \(e\). The restriction in \(\lambda^U\) apparently arose from analogy to precursory systems that used explicitly level-annotated terms [31], but it turns out to be unnecessary with stratified terms.

3.2.1 Provable Equality and Reduction

Figure 3.3 shows the derivation rules for primitive reduction \((\rightarrow_{\text{prim}})\), non-deterministic
reduction (\(\rightarrow\)), and provable equality (\(\doteq\)). Primitive reduction covers mostly the same redex contractions as primitive small-step, but with the key difference that the \(E_V\) rule is generalized to arbitrary levels. Reduction and provable equality are the context- and context-reflexive-symmetric-transitive closures, respectively, of primitive reduction.

Informally, provable equality allows rewriting an arbitrary subterm by either forward or backward primitive reduction. Reduction is the directed counterpart of provable equality that retains the sense of direction of execution; if \(e \rightarrow t\) then \(t\) has the feel of being closer to termination. For example, reduction always preserves the level of a term, which means that reduction only resolves escapes and never introduces them. More generally, a reduct of a value is always a value because reduction never creates new work (although \(\beta\) reduction can replicate existing work).
Lemma 3.11. If $e \in E^0$ then $t \in E^\ell \iff [e/x]t \in E^\ell$.

Proof. Straightforward induction on $t$. \qed

Lemma 3.12 (Level Preservation). $e^\ell \longrightarrow^* t \Longrightarrow t \in E^\ell$.

Proof. Straightforward induction, first on the length of the reduction, then on the first reduction, using Lemma 3.11. \qed

Lemma 3.13. For any $v \in V^\ell$, $v \longrightarrow^* e \Longrightarrow e \in V^\ell$.

Proof. By induction on the length of the reduction $v \longrightarrow^* e$, it suffices to prove this assertion for one-step reductions. The case $\ell > 0$ is just level preservation (Lemma 3.12). For $\ell = 0$, suppose $v$ is $\lambda x.t$ or $\langle t \rangle$ for some $t \in E^0$. Then $e$ is $\lambda x.d$ or $\langle d \rangle$, respectively, where $t \longrightarrow^* d$. By level preservation $d \in E^0$ so $e \in V^0$. \qed

Reduction semantics is directly relevant to reasoning about provable equality because of the Church-Rosser property, which states that expressions are provably equal iff they have a common reduct.

Theorem 3.14 (Church-Rosser Property). $e \doteq t \iff \exists d. e \longrightarrow^* d \longleftrightarrow^* t$.

The Church-Rosser property ensures that reduction is as strong as provable equality itself. Proofs of facts about the equational theory can therefore work with the simpler reduction relation without loss of generality. One important consequence is that reduction and provable equality have the same strength in discovering values to which expressions terminate. This fact makes provable equality easier to connect with small-steps because the correspondence between reduction and small-steps is more direct.

Proposition 3.15. If $e \in E^\ell$ and $v \in V^\ell$ then $e \doteq v \iff \exists u \in V^\ell. e \longrightarrow^* u \doteq v$. 
Proof. For the forward direction, the Church-Rosser property guarantees that \( e \rightarrow^* v \) \( \implies \exists t \in E^\ell. e \rightarrow^* t \leftarrow^* v \) hence \( e \rightarrow^* t \equiv v \). The only question that remains is whether \( t \in V^\ell \), but \( t \) is a reduct of a value so it must be a value as well by Lemma 3.13. The converse follows from \((\rightarrow^*) \subseteq (\equiv)\).  

An equivalent statement to Church-Rosser that is easier to derive is that reducts of a common expression converge via reduction. In this thesis I will prove confluence and take the Church-Rosser property for granted. Proofs that Church-Rosser and confluence imply each other are found in introductory textbooks on rewrite systems or on programming language metatheory \([23, 2]\). The following proof of confluence is known as the Tait-Martin-Löf method, which is explained and used in a number of places \([29, 3, 34, 31]\). The stylized formulation used here is due to Takahashi \([34]\) and is the one used in Taha’s dissertation \([31]\).

**Theorem 3.16 (Confluence).** \( t_1 \leftarrow^* e \rightarrow^* t_2 \implies \exists d. t_1 \rightarrow^* d \leftarrow^* t_2 \).

The main idea is to restate the theorem in terms of an equivalent notion of reduction called parallel reduction, defined in Figure 3.4. Proving confluence for parallel reduction is relatively easy, and this proof directly implies confluence for ordinary reduction by the equivalence of the two notions. A parallel reduction simultaneously reduces a subset of the redexes in the original term. The superscript on \((\gg)\) is the parallel reduction’s complexity, which is the number of ordinary reductions it would take to mimic the parallel reduction in a leftmost-outermost-first manner. This information is used later to prove equivalence with deterministic semantics; for proving confluence I discard the complexity annotation and work with \((\gg)\).

**Lemma 3.17.** \((\gg)^*\) = \((\rightarrow^*)\).
Figure 3.4: Parallel reduction.

Proof. Derivation rules for \((\gg^*)\) subsume primitive reduction rules and RED-Ctx, so \((\rightarrow^*) \subseteq (\gg^*)\). For the reverse containment, \(e \gg t \Rightarrow e \rightarrow^* t\) by straightforward induction on the parallel reduction judgment. Therefore, \(e \gg^n t \Rightarrow e \rightarrow^* t\) by induction on \(n\). □

Thus for Theorem 3.16 it suffices to prove that \((\gg)\) is confluent. Intuitively, a parallel reduction \(e \gg t\) reduces some subset of all independent redexes in \(e\), so by reducing a complementary set of redexes, the \(t\) reduces to a form \(e^*\) that depends only on \(e\) and is independent of \(t\) (Takahashi’s property). The \(e^*\) is called the complete development and is formalized in Figure 3.5. Then given a pair of departing reductions \(t \ll^* e \gg^* d\), reductions on the left branch can be connected to the right branch by
\[
\begin{align*}
  x^* & \overset{\text{def}}{=} x \\
  (\lambda x.e)^* & \overset{\text{def}}{=} \lambda x.e^* \\
  ((\lambda x.e^0) a)^* & \overset{\text{def}}{=} [a^*/x](e^0)^* \\
  \langle e \rangle^* & \overset{\text{def}}{=} \langle e^* \rangle \\
  (-e)^* & \overset{\text{def}}{=} -(e^*) \quad [\text{if } e \neq \langle e' \rangle] \\
  \text{run } e^* & \overset{\text{def}}{=} \text{run } e^* \quad [\text{if } e \neq \langle e^0 \rangle]
\end{align*}
\]

Figure 3.5: Complete development.

\[e^* \text{ is the complete development of } e.\]

repeatedly taking complete developments.

**Lemma 3.18** (Takahashi’s Property). \( e \gg t \Longrightarrow t \gg e^* \).

**Proof.** Induction on \( e \) using Lemmas 3.12 and 3.13. See Section A.2 for details. \(\square\)

**Proposition 3.19.** \((\gg^*)\) is confluent.

**Proof.** This assertion reduces to Takahashi’s property by lexicographical induction on the lengths of the departing parallel reductions (i.e. given \( t_1 \ll^n e \gg^m t_2 \), induct on \((n,m)\)). See Section A.2 for details. \(\square\)

### 3.2.2 Compatibility With Evaluation

This subsection proves Plotkin-style correspondence between the equational theory and the small-step semantics. The main milestone in the proof is that an expression is equal to a value iff the expression terminates by small-steps; soundness then follows directly. Completeness does not hold for provable equality. Generally, a sound axiomatic equational theory for \((\approx)\) based on decidable pattern-matching, such as \((\doteq)\),
is incomplete because an inductively defined relation with decidable derivation rules is semi-decidable whereas \((\approx)\) is not semi-decidable for a Turing-complete language.

**Theorem 3.20** (Equivalence of Axiomatic and Small-step Semantics). If \(e \in E^\ell\) and \(v \in V^\ell\) then \(e \doteq v \iff \exists u \in V^\ell.\ e \sim^*_\ell u \doteq v\).

**Corollary 3.21** (Soundness and Incompleteness). \((\doteq) \subset (\approx)\).

**Proof.** Let \(e, t, C\) be given such that \(e \doteq t\) and \(C[e], C[t] \in E^0\). Let us suppose that one of the plugged expressions terminates, say \(C[e] \Downarrow^0\), and prove that the other also does. By definition, \(\exists v \in V^0.\ C[e] \leadsto^* v\) so using Theorem 3.20 and EQ-Ctx, \(v \doteq C[e] \doteq C[t]\). Then by Theorem 3.20 again, \(C[t] \Downarrow^0\). The containment is proper because \((\approx)\) is not semi-decidable (since \(\lambda^V\) is clearly a superset of the Turing-complete \(\lambda\) calculus), whereas \((\doteq)\) is clearly computationally enumerable. See Section 5.1 for concrete examples of equivalences in \((\approx) \setminus (\doteq)\). \(\square\)

As with confluence, the proof of Theorem 3.20 is more or less the same as for \(\lambda^U\) [31], which follows Takahashi’s approach [34]. The first step is to convert the theorem to a statement about \((\leadsto)\) instead of \((\doteq)\). This conversion is justified by Proposition 3.15.

**Theorem 3.22** (Equivalence of Reduction and Small-step Semantics). If \(e \in E^\ell\) and \(v \in V^\ell\) then \(e \longrightarrow^* v \iff \exists u \in V^\ell.\ e \sim^*_\ell u \longrightarrow^* v\).

**Proof of Theorem 3.20 Assuming Theorem 3.22.**

\((\implies)\) Suppose \(e \doteq v\). Then by Proposition 3.15, \(\exists w \in V^\ell\) such that \(e \longrightarrow^* w \doteq v\).

By Theorem 3.22, \(\exists u \in V^\ell.\ e \sim^*_\ell u \longrightarrow^* w\) so \(e \sim^*_\ell u \doteq w \doteq v\).

\((\impliedby)\) This direction doesn’t actually need Theorem 3.22. If \(e \sim^*_\ell u \doteq v\), then \(e \doteq u \doteq v\) because \((\sim) \subseteq (\longrightarrow) \subseteq (\doteq)\). \(\square\)
Hence I can focus on Theorem 3.22 instead of Theorem 3.20. The \((\iff)\) direction in Theorem 3.22 is trivial. The \((\implies)\) direction is proved via three lemmas that convert a parallel-reduction sequence

\[ e_0 \gg e_1 \gg \cdots \gg e_n = v^\ell \]

to a small-step sequence

\[ e_0^\ell \leadsto t_1^\ell \leadsto t_2^\ell \leadsto \cdots \leadsto t_m^\ell = u^\ell \gg v^\ell. \]

**Lemma 3.23** (Transition). If \(e \in E^\ell\) and \(v \in V^\ell\) then \(e \gg^n v \implies \exists u \in V^\ell. e \leadsto^*_\ell u \gg v.\)

*Proof.* If \(e \in V^n\), then just take \(u \overset{\text{def}}{=} e\). Otherwise, induct on \((n,e)\) under the lexicographical ordering, with case analysis on the last rule used to derive the parallel reduction. The proof uses Lemmas 3.12 and 3.13. See Section A.3 for details. \(\square\)

**Lemma 3.24** (Permutation). If \(e,t,d \in E^\ell\) then \(e \gg^n t \leadsto d \implies \exists t' \in E^\ell. e \leadsto^{+^\ell} t' \gg d.\)

*Proof.* Induction on \(n\) with case analysis on the last rule used to derive the parallel reduction, using Lemmas 3.12 and 3.13. See Section A.3 for details. \(\square\)

**Lemma 3.25** (Push Back). If \(e,t \in E^\ell\) and \(v \in V^\ell\) then \(e \gg t \leadsto v \implies \exists u \in V^\ell. e \leadsto^{+^\ell} u \gg v.\)

*Proof.* Let the length of the small-step sequence be \(n\). Induct on \(n\).

[If \(n = 1\)] By Permutation \(\exists t' \in E^\ell. e \leadsto^+ t' \gg v\), so by Transition, \(\exists u \in V^\ell. t' \leadsto^*_\ell u \gg v\). Putting them together, \(e \leadsto^+ t' \leadsto^*_\ell u \gg v\).

[If \(n > 1\)] By hypothesis, \(\exists d. e \gg t \leadsto d \leadsto^{(n-1)} (n-1) v\). Permutation gives \(\exists t'. e \leadsto t' \gg d \leadsto^{(n-1)} (n-1) v\). Then by IH \(\exists u \in V^\ell. t' \leadsto^+ u \gg v\), so \(e \leadsto t' \leadsto^+ u \gg v\). \(\square\)
Proof of Theorem 3.22.

(\imp) If \( e \xrightarrow{*} v \) then \( e \gg^n v \) for some \( n \) by Lemma 3.17. I wish to show
\[
\exists u \in V^\ell. \exists m \geq 0. e \xrightarrow{\ell}^m u \gg v \text{ by induction on } n. \text{ If } n = 0 \text{ then } u \overset{\text{def}}{=} v. \text{ If } n > 0 \\
\text{then } \exists t \in E^\ell. e \gg t \gg^{n-1} v \text{ so by IH } \exists u \in V^\ell. \exists m' \geq 0. e \gg t \xrightarrow{\ell}^m' u \gg^* v. \text{ Then}
\]
the conclusion follows from Transition if \( m' = 0 \), or from Push Back if \( m' > 0 \).

(\eqv) Follows from \( (\over) \subseteq (\imp) \).

3.3 Anti-Equational Theory: What Are Not Equal

In this section, I explain why certain tempting extensions to the theory are unsound. Two main focuses are dropping level constraints from primitive reduction rules and treatment of variables as values (or rather, as substitutables).

3.3.1 Generalized Reduction Axioms

The reduction/equational axioms of \( \lambda^V \) are derived from \( \lambda^U \) by generalizing the RED-E\_V axiom. The counterpart in \( \lambda^U \), named \( E^U \), states \( \tilde{\langle} e^0 \tilde{\rangle} \xrightarrow{\text{prim}} e^0 \) whereas \( E^V \) is applicable at arbitrary levels: \( \tilde{\langle} e \tilde{\rangle} \xrightarrow{\text{prim}} e. \) This is a substantial generalization that I will need in Chapter 5. It is natural to wonder whether similar generalizations are possible for the two other rules, RED-\( \beta \) and RED-\( R_V \). Unfortunately, generalizing the other rules in the same manner leads to unsoundness.

My observations here are an extension to Taha’s insights on the pathologies in earlier prototypes of the staged \( \lambda \) calculus. As such, some of the examples are directly drawn from his dissertation [31]. The contribution of my analysis here is to confirm that the same problems arise in \( \lambda^V \) and to provide a more systematic, thorough walk-through of the equational theory’s design space. In particular, I offer a clear
explanation why RED-\(E_V\) could be generalized whereas RED-\(\beta\) and RED-\(R_V\) could not. I formalize the intuitive reason behind the failure of most of these generalizations and show that it provides a useful sanity check for any new equational rule, should one be proposed.

Let \(\Omega\) be a closed, divergent level-0 expression. First consider generalization to RED-\(R_V\). If RED-\(R_V\) is generalized to arbitrary levels, so that

\[
\text{run} \langle e \rangle \xrightarrow{\text{prim}} e,
\]

then the reduction semantics proves

\[
\langle \text{run} \langle \tilde{\Omega} \rangle \rangle = \langle \tilde{\Omega} \rangle
\]

but the left-hand side is a level-0 value while the right-hand side diverges at level 0, so the associated equational theory is unsound.

For RED-\(\beta\), both the function body and the argument are potential targets for generalization. I will consider each of them in turn—I will take the argument first. The options for how to generalize the argument naturally differ between evaluation strategies. For CBN, generalizing the argument level yields the rule

\[
(\lambda x.e^0) t \xrightarrow{\text{prim}_n} [t/x]e^0.
\]

For CBV, a naïve generalization is absurd:

\[
\forall \ell. (\lambda x.e^0) v^\ell \xrightarrow{\text{prim}_v} [v^\ell/x]e^0.
\]

As \(V^1 = E^0\), this rule essentially permits CBN reduction, which is hopelessly unsound. A more reasonable approach would be to retain the constraint on the head constructor.

\[
(\lambda x.e^0) (\lambda y.t) \xrightarrow{\text{prim}_v} [(\lambda y.t)/x]e^0
\]

\[
(\lambda x.e^0) \langle t \rangle \xrightarrow{\text{prim}_v} [(t)/x]e^0
\]
Unfortunately, none of these four rules is sound. Every one of them except (∗5) proves
\[ \langle \langle \lambda x. \langle x \rangle \rangle \langle \lambda y. \Omega \rangle \rangle \triangleright \langle \langle \lambda y. \Omega \rangle \rangle \]  
(∗6)
and (∗5) proves
\[ \langle \langle \lambda x. \langle x \rangle \rangle \rangle \langle \tilde{\Omega} \rangle \rangle \triangleright \langle \langle \tilde{\Omega} \rangle \rangle \]  
(∗7)
where I used teletype font for the parentheses to make them better distinguishable from staging brackets. In both of these counterexamples, the left-hand sides diverge while the right-hand sides terminate.

Generalizing the abstraction body also leads to unsoundness. With the generalization
\[ (\lambda x.e) a \rightarrow_{\text{prim}} [a/x]e, \]  
(∗8)
the equational theory proves
\[ \langle (\lambda x.\tilde{x}) \langle e^0 \rangle \rangle \triangleright \langle \tilde{\langle e^0 \rangle} \rangle. \]  
(∗9)
The left-hand side is divergent (stuck) while the right-hand side terminates in one step at level 0, so this generalization is unsound.

The root cause of failure for generalized run (∗1) and argument-generalized β (∗3)(∗4)(∗5) is a mismatch in the contextual level on both sides of the rewrite rule. For example, in the rewrite rule (∗1), the expression \( e \) appears inside one more pair of brackets on the left than on the right. An \( e \) that contains divergence enclosed in the right number of escapes can then cause one side to diverge but not the other, as in (∗2)(∗6)(∗7). This idea can be formalized and generalized to other potential rewrite rules, as follows.

**Definition 3.26 (Exact Level).** Define \( \text{lv} : E \rightarrow \mathbb{N} \) as
\[
\begin{align*}
\text{lv} x & \overset{\text{def}}{=} 0 \\
\text{lv} (e_1 e_2) & \overset{\text{def}}{=} \max(\text{lv} e_1, \text{lv} e_2) \\
\text{lv}(\tilde{e}) & \overset{\text{def}}{=} \text{lv} e + 1 \\
\text{lv}(\lambda x.e) & \overset{\text{def}}{=} \text{lv} e \\
\text{lv}(\langle e \rangle) & \overset{\text{def}}{=} \max(\text{lv} e - 1, 0) \\
\text{lv} \langle\text{run } e\rangle & \overset{\text{def}}{=} \text{lv} e
\end{align*}
\]
This definition is called the exact level of $e$ because $\ell v\ e$ returns the exact nesting depth of escapes in $e$, whereas the assertion $e \in E^\ell$ only gives an upper bound.

**Proposition 3.27.** The exact level $\ell v\ e$ is the least $\ell$ for which $e \in E^\ell$, or equivalently, the unique $\ell$ for which $e \in E^\ell \setminus E^{\ell-1}$ where $E^{-1} \equiv \emptyset$ when $\ell = 0$.

**Proof.** Straightforward induction on $e$. \hfill \Box

**Definition 3.28 (Level Function).** Define $\Delta : Ctx \to \mathbb{Z}$ as follows.

$$
\Delta \bullet \overset{\text{def}}{=} 0 \quad \Delta(C\ e) \overset{\text{def}}{=} \Delta C \quad \Delta(C) \overset{\text{def}}{=} \Delta C - 1 \quad \Delta(\text{run } C) \overset{\text{def}}{=} \Delta C
$$

$$
\Delta(\lambda x. C) \overset{\text{def}}{=} \Delta C \quad \Delta(e\ C) \overset{\text{def}}{=} \Delta C \quad \Delta(\neg C) \overset{\text{def}}{=} \Delta C + 1
$$

The level function $\Delta C$ is an integer difference that captures the limiting behavior of the function $e \mapsto \ell v C[e] - \ell v e$ as $\ell v e \to \infty$. Intuitively, the function $\ell v C[e] - \ell v e$ converges to a constant because when $e$ is sufficiently high-level, the deepest nesting of escapes in $C[e]$ occurs within $e$. Then the exact level of the whole term $C[e]$ is determined by the exact level of $e$ and the number of escapes and brackets in $C$ that enclose the hole. The latter number is just $\ell v C[e] - \ell v e$.

**Proposition 3.29.** $\forall C. \exists \ell(C) \in \mathbb{N}. \ \ell v e \geq \ell(C) \implies \ell v C[e] = \ell v e + \Delta C$.

**Proof.** Induction on $C$. See Section A.4 for details. \hfill \Box

With these definitions in place, the general claim is that a rewrite rule which moves an arbitrary expression from one context to another is unsound unless the contexts have the same level function. That the level function is the limit of $\ell v C[e] - \ell v e$ is essential here. A sound rule can move an expression between contexts $C$ and $C'$ such that $\ell v C[e]$ and $\ell v C'[e]$ disagree for some $e$, as long as those $e$ are all low-level. For example, RED-$E_V$ rewrites $\langle e \rangle$ to $e$, but if $e \in E^0$ then $\ell v \langle e \rangle = 1 \neq \ell v e$ which is
to say $\text{lv } \langle e \rangle - \text{lv } e \neq \text{lv } e - \text{lv } e$. The exact levels however agree whenever $\text{lv } e \geq 1$, which is why the following proposition does not apply to RED-$E_V$.

**Proposition 3.30.** Any rewrite rule that has the form or subsumes a rule of the form $C[e] \rightarrow C'[e]$ with $\Delta C \neq \Delta C'$ is unsound ($\exists e. C[e] \not\approx C'[e]$).

The proof of this proposition relies on the fact that if $e$ has enough escapes, it can dominate all the staging annotations in $C$ and be given top priority during program execution. In more technical terms, $\text{lv } C[e]$ grows unboundedly with $\text{lv } e$ because of Proposition 3.29, and beyond a certain threshold $C \in \text{ECtx}^{\ell,\ell - \Delta C}$. Then by Lemma 3.6, $e$ is evaluated first, which can diverge in $C$ but not in $C'$ (or vice versa). Notice that this proof fails, as expected, if the $e$ in $C[e] \rightarrow C'[e]$ is restricted to $e^0$.

**Lemma 3.31** (Context Domination). $\text{size}(C) < \ell \implies \exists m. C \in \text{ECtx}^{\ell,m}$.

**Proof.** Induction on $C$. See Section A.4 for details.

**Lemma 3.32.** $\Delta \mathcal{E}^{\ell,m} = \ell - m$.

**Proof.** Straightforward induction on $\mathcal{E}^{\ell,m}$.

**Proof of Proposition 3.30.** Take $\ell \overset{\text{def}}{=} \max(L(C), L(C'), \text{size}(C) + 1, \text{size}(C') + 1)$ and $e = \underbrace{\ldots \ldots \underbrace{\Omega, \ldots}}_{\ell \text{ times}},$ where $\Omega \in E^0$ and $\Omega \uparrow^0$. Then $\text{lv } e = \ell$, $e \uparrow^\ell$, $\text{lv } C[e] = \ell + \Delta C$, and $\text{lv } C'[e] = \ell + \Delta C'$. Without loss of generality, $\Delta C > \Delta C'$. By Lemma 3.31, $C \in \text{ECtx}^{\ell + \Delta C, \ell}$ where the second superscript is known by Lemma 3.32. Then taking $C_{\langle \ldots \rangle} \overset{\text{def}}{=} \langle \cdots \langle \bullet \cdots \rangle \cdots \rangle$ with $\ell + \Delta C$ pairs of brackets, $C_{\langle \ldots \rangle}[C] \in \text{ECtx}^{0, \ell}$, so Lemma 3.6 forces $C_{\langle \ldots \rangle}[C[e]] \uparrow^0$. By contrast, $\text{lv } C'[e] < \ell + \Delta C$, so $C_{\langle \ldots \rangle}[C'[e]]$ is of the form $\langle d^0 \rangle$, hence $C_{\langle \ldots \rangle}[C'[e]] \downarrow^0$. \qed
Of course, for the contexts to have the same level function (or for the expression being moved around to be restricted to level 0) is just a necessary condition for soundness and not a sufficient condition. For example, the rule

\[
\langle \neg e \rangle \rightarrow e
\]

passes the level function test but is unsound because if \( e = \lambda x.t^0 \) then \( \langle \neg \lambda x.t^0 \rangle \uparrow^0 \) but \( e \downarrow^0 \). Nonetheless, the level function is a useful tool that help to form reasonable conjectures and to quickly discard unreasonable ones. For the rule just presented, the fact that it passes the level function test suggests that this rule may be fixable, at least more likely so than rules that do not. Indeed, checking that \( e \) terminates to a code value (if it terminates at all) is sufficient to make this rule sound:

\[
\text{run } \langle \neg e \rangle \rightarrow \text{run } e.
\]  

(*10)

The soundness of this rule is not directly derivable in the equational theory presented in this chapter. I will prove this rule’s soundness in Chapter 5, as Example 5.9.

The abstraction body-generalized \( \beta \) rule is of a different nature. Although the level functions do mismatch in this rule, the expression being moved is restricted to level 0, so Proposition 3.30 does not apply. Indeed, the counterexample (*9) is not of the form described in the proof of Proposition 3.30. The cause of unsoundness for this rule is dual to the problem in Proposition 3.30: substitution injects an extra pair of brackets at a location that was previously stuck on an ill-leveled variable, whereas Proposition 3.30 injects extra escapes. The ill-leveled nature of \( x \) is essential here. The extra brackets only help termination if the variable being substituted for used to be the source of stuck execution, which only happens if the variable appears in a level-0 context. But execution never goes under binders at level 0, so this variable must have been bound at level > 0. I therefore conjecture that in a type-safe system,
generalization of the function body is sound. I do not have a rigorous proof of this conjecture, however.

3.3.2 Substitution of Variables for Variables

So far, the metatheory of $\lambda^V$ has mostly mirrored that of $\lambda^n$ without significant differences, including the proofs of confluence and soundness. However, $\lambda^V$ has an annoying idiosyncrasy that hampers seemingly straightforward equational reasoning, and that does not arise in $\lambda^n$ nor in plain $\lambda_v$. Namely, as the careful reader may have noticed in Figure 3.2, a variable is not a CBV substitutable, and an application cannot be contracted if the argument is just a variable. This subsection explains intuitively why only $\lambda^V$ has this limitation and proposes a general workaround.

Traditionally, CBV calculi admit the equational rule

$$\lambda y.e^0 \quad \vdash [x/y]e^0 \quad \text{[EQ-Var}_\beta\text{]}$$

Plotkin’s seminal $\lambda_v$ calculus [29], for example, does so implicitly by taking variables to be values. EQ-$\text{Var}_\beta$, however, is not admissible in $\lambda^V$. This rule does not exist in $\lambda^V$ either as an axiom or a derived rule, and adding the rule makes the equational theory unsound. Consider the expression $(\lambda x.(\langle y \rangle x))$ and its $\text{Var}_\beta$ reduct $\langle y \rangle$. Replacing one of these by the other appears deceptively benign, but the two terms are actually nonequivalent as shown by the distinguishing context $\langle \lambda x. \cdot \rangle$:

$$\langle \lambda x. \cdot ((\lambda x.(\langle y \rangle x)) \uparrow^0 \quad \langle \cdot \rangle \downarrow^0 \langle y \rangle$$

Formally, the expression on the left diverges because $x \notin A_v$. Practically, a typical implementation of untyped MSP would report an unbound variable error upon seeing the $x$. Notice that the pathology only surfaces when $x$ is ill-leveled.

I should remark that there does exist a way to make EQ-$\text{Var}_\beta$ admissible. If the
small-step semantics acquires the rule

\[ (\lambda x.e^0) \ x^0 \xrightarrow{\text{prin}_0} [y/x]e^0 \]

then EQ-VAR\(\beta\) is sound with respect to this modified semantics. The associated reduction semantics remains confluent, and the equational system will be better-behaved. Adopting SS-VAR\(\beta\), however, is inadvisable for practical purposes. Implementing SS-VAR\(\beta\) requires a runtime representation for variables which can be expensive and is probably of little practical use. In fact, implementing the SS-VAR\(\beta\) rule can be harmful. The rule applies only when the variable in argument position has an ill-leveled binding, which most likely indicates a programmer error. The more helpful course of action, then, is to report an error and not to carry on with the execution; consequently, the correct choice in the \(\lambda V\) calculus is to reject EQ-VAR\(\beta\), which is the choice made in this thesis.

The problem with EQ-VAR\(\beta\) is unique to the CBV untyped multi-stage setting. Untyped \(\lambda V\) is free from this limitation because CBN RED-\(\beta\) subsumes EQ-VAR\(\beta\) because \(A_n = E^0 \supseteq \text{Var}\). A typed refinement of \(\lambda V\) can admit SS-VAR\(\beta\) as well, provided that it is type-safe—no well-typed program gets stuck—with respect to the semantics without SS-VAR\(\beta\). In such an extension, the type system prevents a VAR\(\beta\) redex \((\lambda x.e)\ y\) from ever occupying the hole of the evaluation context during the execution of a program. Thus a typed system whose underlying untyped semantics includes SS-VAR\(\beta\) is indistinguishable from a typed system that is laid over an un-

*Not to be confused with the notion of symbols featured in LISP. A symbol is always a quoted datum in LISP (otherwise it is a variable lookup), which is analogous to being enclosed in brackets.

†I mean uniqueness here among the three-dimensional semantic variations: CBV vs CBN, multi-stage vs single-stage, and typed vs untyped. Some language features other than MSP may also introduce this problem, e.g. mutable variables [11].
typed semantics that excludes SS-VAR$\beta$, and the former can be directly proved to permit EQ-VAR$\beta$.

A general solution to this problem that does not resort to a type system or to CBN exists. Namely, EQ-VAR$\beta$ can recover soundness by slightly restricting its form:

\[
\frac{C[(\lambda y.e^0)\; x] \in E^0 \quad C \text{ does not bind } x}{\lambda x.C[(\lambda y.e^0)\; x] \approx_{v} \lambda x.C[[x/y]e^0]} \quad \text{[EQ-VAR$\beta$]}
\]

Intuitively, the problem with EQ-VAR$\beta$ is that given only an expression $(\lambda y.e^0)\; x$, there is no way to tell whether $x$ is guaranteed to be well-leveled when the expression is put back into context. The EQ-VAR$\beta$ rule demands this guarantee by requiring the binder to be visible and at level 0. The soundness of EQ-VAR$\beta$ naturally falls out of the extensionality principle in Chapter 5, so I will defer the proof until then.
Chapter 4

Effects of Staging on Correctness

In the preceding chapter, I proved that familiar reasoning principles transfer from the plain λ calculus to λ^V. This chapter presents techniques for using these principles to reason about specific staged programs. The techniques are based on the Erasure Theorem, which captures the effects of staging annotations on the behavior of a program. An erasure ||e|| is the expression e with all staging constructs stripped away; the erasure theorem states that erasure preserves provable equality, or equivalently preserves reduction, which implies ||e|| = e under certain simple conditions. These conditions constitute the Correctness Criteria for staging.

The motivation behind erasure is that many MSP applications are like partial evaluation (PE) in that it is derived from an unstaged program by adding staging annotations to that program without changing the overall algorithm. The original program is said to be the unstaged counterpart of the staged program. The gibst example in Section 2.2 is PE-like; it is derived from the unstaged gib by simply adding staging annotations without changing the program logic. Erasure formalizes this idea of an unstaged counterpart. A PE-like staged program (or expression), then, is one that is intended to behave identically to the erasure.

The statement of the Erasure Theorem differs for CBV and CBN. The CBN version states that reductions of a staged term e directly map to legitimate reductions of the erasure ||e||. As a result, the staged program is equal to the unstaged program if all staging annotations can be resolved via reductions and reverse reductions. The
CBV version of the theorem is similar but has a caveat that erasure does not map CBV reductions to CBV reductions, but to reductions which are in general CBN (rather than CBV). The reason for this mismatch in evaluation strategy is that brackets can protect an argument from divergence and therefore justify CBV $\beta$ substitution, but in the erasure of such a term the protection is gone and the argument diverges. There are two approaches to overcoming this limitation, both of which I explain in this chapter. One is to prove that the difference in evaluation strategy is irrelevant, and the other is to restrict the form of equality so that the erasure is guaranteed to fall in CBV.

By these results, the programmer can focus on analyzing erasures of staged programs, for which traditional analysis techniques such as those developed in Chapters 3 and 5 work better. One caveat in this proof method is that staging a curried function can cause it to diverge prematurely upon partial application whereas its erasure does not, even if the functionality of the staged and unstaged counterparts are essentially the same. This problem is easy to solve but is quite pervasive and almost always needs attention; see Section 4.3 for an example.

### 4.1 The Erasure Theorem

**Definition 4.1 (Erasure).** Define the erasure $\|e\|$ of an expression $e$ by

\[
\begin{align*}
\|x\| &\overset{\text{def}}{=} x \\
\|\lambda x.e\| &\overset{\text{def}}{=} \lambda x.\|e\| \\
\|\tilde{e}\| &\overset{\text{def}}{=} \|e\| \\
\|e_1 e_2\| &\overset{\text{def}}{=} \|e_1\| \|e_2\| \\
\|\langle e \rangle\| &\overset{\text{def}}{=} \|e\| \\
\|\text{run } e\| &\overset{\text{def}}{=} \|e\|
\end{align*}
\]

The $\|\cdot\|$ operation produces annotation-free terms and is idempotent. A few more facts about erasure follow, which should be intuitively clear. All of these facts, along with the very definition of erasure, is common to both CBN and CBV.
Lemma 4.2. $e \in E^t \implies \|e\| \in E^t$.

Proof. Straightforward induction on $e$. \qed

Lemma 4.3. $\|[t/x]\|e\| = \|[t/x]e\|$.

Proof. Straightforward induction on $e$. \qed

Lemma 4.4. If $\|e\| \to t$ then $t = \|t\|$.

Proof. Straightforward induction on the reduction judgment using Lemma 4.3. \qed

Formally, the erasure $\|e\|$ is the unstaged counterpart of the staged expression $e$. The goal then is to prove $e \approx \|e\|$. In general, such a proof can involve a full-blown analysis of the execution of $e$ and $\|e\|$ under arbitrary contexts, but the Erasure Theorem simplifies the proof in common cases.

The intuition behind the Erasure Theorem is that all that staging annotations do is to describe and enforce a particular evaluation strategy. They may force CBV, CBN, or some strategy in between that the programmer believes is most efficient for the program at hand. But in CBN, a reduction ($\to_n$) is already capable of reducing by any reduction strategy—the redex can be chosen anywhere. It follows that erasure preserves CBN reductions. In this sense, erasure is a monotone function from the partial order $(E, \to^*_n)$ to itself, or equivalently, an endomorphism on the algebra $(E, \to_n)$.

Remark 4.5. The observation made here does not show that staging is useless in CBN. Recall that CBN reduction is really an equality in disguise and not a semantics specifying how to evaluate programs. Being an equality allows CBN reduction to contract redexes at arbitrary locations and thereby trace arbitrary evaluation strategies.
CBN small-step, by contrast, follows a fixed evaluation strategy, so there are performance benefits to staging annotations which can enforce any evaluation strategy on the small-steps.

A similar conclusion can be drawn for CBV reduction, but with a twist. Erasing a CBV reduction does not always result in a valid CBV reduction but in a reduction that is in general CBN. This discrepancy arises because CBV reduction is a strict subset of CBN reduction, and some evaluation strategies are not traceable by CBV reduction. Staging annotations can enforce any evaluation strategy, including ones that are not traceable by CBV reduction, so the best that can be said about the erasures of staged CBV reductions is that they are valid CBN reductions. Thus erasure is a monotone function from \((E, \rightarrow_v^n)\) to \((E, \rightarrow_n)\) and a homomorphism from \((E, .\ = v)\) to \((E, .\ = n)\).

**Theorem 4.6** (Erasure). Erasure preserves CBN reduction: if \(e \rightarrow^n t\) then \(\|e\| \rightarrow^n \|t\|\). Erasure converts CBV reduction to CBN reduction: if \(e \rightarrow^*_v t\) then \(\|e\| \rightarrow^n \|t\|\).

**Proof.**

[CBN] By induction on the length of the reduction, I only need to handle the case where \(e \rightarrow^n t\). Decomposing this reduction as \(C[r] \rightarrow^n C[d]\), all that needs to be shown is \(\|r\| \rightarrow^n \|d\|\), for then \(\|C[r]\| = (\|C\|)[\|r\|] \rightarrow^n (\|C\|)[\|d\|] = \|C[d]\|\).

[If \(r = \langle d \rangle\) or \(\langle \text{run } d \rangle\)] \(\|r\| = \|d\|\).

[If \(r = (\lambda x.e_1) e_2\) for some \(e_1, e_2 \in E^0\)] \(\|r\| = (\lambda x.\|e_1\|) \|e_2\| \rightarrow^n \|e_2\|/x\|e_1\| = (\ast 1)\), \(\|e_1/x\|e_2 = \|d\|\), where \((\ast 1)\) is by Lemma 4.2 and \((\ast 2)\) is by Lemma 4.3.

Therefore, \(\|\cdot\|\) preserves CBN reduction.
[CBV] The CBV statement follows from the CBN statement because \((\rightarrow^\nu) \subseteq (\rightarrow^\ell_n)\).

A qualitative reading of this theorem is that, because of confluence, annotations preserve the return value of a program modulo CBN provable equality and modulo leftover annotations in the return value. Only CBN equality is guaranteed even in CBV. This result answers one of the questions raised in Section 2.3, namely, annotations can affect the termination of a program but not its return value.

**Corollary 4.7.** If \(v, u \in V^\ell\), then \((e \rightarrow^* v \land \|e\| \rightarrow^* u) \implies \|v\| \doteq_n u\). Note the conclusion uses CBN equality even if the premise uses CBV reduction.

*Proof.* By the Erasure Theorem, \(\|e\| \rightarrow^* \|v\|\) so \(\|v\| \doteq_n \|e\| \doteq_n u\).

In CBN, the Erasure Theorem further guarantees that staging can never make programs more terminating. This property was stated as the answer to one of the other questions raised in Section 2.3.

**Corollary 4.8.** If \(e \downarrow\|v\|\), then \(\|e\| \downarrow\|v\|\).

*Proof.* By Theorem 3.22, \(e \downarrow\|v\|\) implies \(e \rightarrow^* \|v\|\). By the Erasure Theorem, \(\|e\| \rightarrow^* \|\|v\|\| = \|v\|\), using the idempotence of \(\|\cdot\|\).

A slightly stronger corollary is the following, which establishes a correctness criterion for PE-like programs. It is not as straightforward to interpret as the two preceding corollaries, but it is more useful because the premise is easier to prove. The corollary states that if \(e\) is provably equal to any unstaged term, then \(e \doteq_n \|e\|\). Once again, the notion of equality in the conclusion is CBN even if the original equality is CBV. This guarantee is needed to rule out cases where \(e\) gets stuck due to ill-formed uses of staging annotations, for instance \(e = (\lambda x.e^0) \ t^0\).
Corollary 4.9 (General Correctness Criterion). \( \exists t \in E. \ e \doteq ||t|| \implies e \doteq_n ||e|| \). Note that the conclusion is CBN equality even if the premise uses CBV equality.

Proof. For the forward direction, notice that by confluence, \( e \rightarrow^* d \leftarrow^* ||t|| \) for some \( d \). By Lemma 4.4, \( d = ||d|| \) so \( e \rightarrow^*_n ||d|| \). By the Erasure Theorem, \( ||e|| \rightarrow^*_n ||d|| \), therefore \( ||e|| \doteq_n e \).

With this corollary, \( e \doteq_n ||e|| \) can be proved by starting from \( e \) and reaching any unstaged term \( ||t|| \) via reductions and reverse reductions. Intuitively, the premise \( \exists t \in E. \ e \doteq ||t|| \) amounts to ensuring that staging annotations will go away with reductions and reverse reductions. In CBN, finding such a \( t \) concludes the verification, but in CBV there is the extra obligation to justify converting \( e \doteq_n ||e|| \) to \( e \approx_v ||e|| \).

A natural question to ask at this point is what makes the stronger assertion \( e \rightarrow^*_v t \implies ||e|| \rightarrow^*_v ||t|| \) fail and whether CBV reduction can be recovered between the erasures by restricting the reduction in the premise. The main problem with erasing CBV reductions is that the argument in a CBV \( \beta \) reduction may become divergent in the erasure when the brackets had made it a value in the original term. For example, if \( ||\Omega|| \uparrow^0 \) then \((\lambda x.e^0) \langle||\Omega||\rangle \rightarrow^*_v \langle||\Omega||\rangle/x\rangle e^0 \) but \((\lambda x.||e^0||) ||\Omega|| \rightarrow^*_v ||\Omega||/x||e^0|| \).

It can be shown easily that ruling out this case is sufficient to recover CBV reduction between the erasures.

Definition 4.10 (Careful Reduction). Define \( \rightarrow_{\text{vc}} \) by the same rules as \( \rightarrow_v \), except with RED-\( \beta \) restricted to

\[
\frac{||v^0|| \downarrow^0_v}{(\lambda x.e^0) v^0 \rightarrow_{\text{vc}} [v^0/x] e^0} \quad \text{[RED-} \beta_{\text{vc}}]\]

Let \( \doteq_{\text{vc}} \) be its symmetric-reflexive-transitive closure.

Lemma 4.11. If \( t^0 \downarrow^0_v \) then \((\lambda x.e^0) t^0 \doteq_v [t^0/x] e^0 \).
Proof. By assumption $t^0 \vdash v^0$ for some $v^0$, so $(\lambda x.e^0) t^0 \vdash (\lambda x.e^0) v^0 \vdash [v^0/x]e^0 \vdash [t^0/x]e^0$. I omit the straightforward induction needed to justify the last step. \qed

**Theorem 4.12** (Careful Erasure). $e \rightarrow^*_{\text{vc}} t \implies \|e\| \rightarrow^*_{\text{v}} t$. Note the premise uses careful reduction but the conclusion uses ordinary reduction.

Proof. The proof is mostly the same as the CBN variant of the Erasure Theorem: induct on the length of the reduction, then induct on the derivation of the reduction and perform case analysis on the form of the redex. The only difference is, of course, in the handling of $\beta_{\text{vc}}$ reduction.

Let $e = (\lambda x.d') v$ for some $d' \in E^0$ and $v \in V^0$. Because the form of the redex is restricted, $\|v\| \downarrow^0_{\text{v}}$. Therefore, by Lemmas 4.3 and 4.11, $\|e\| = (\lambda x.\|d'\|) \|v\| \vdash_{\text{v}} [\|v\|/x]\|d'\| = \|[v/x]d'\|$. \qed

**Corollary 4.13.** If $v, u \in V^\ell$ then $(e \rightarrow^*_{\text{vc}} v \land \|e\| \rightarrow^*_{\text{v}} u) \implies \|v\| \vdash_{\text{v}} u$. Note the first reduction is careful but the second one is not.

Proof. By the Careful Erasure Theorem, $\|e\| \vdash_{\text{v}} \|v\|$, so $u \vdash_{\text{v}} \|e\| \vdash_{\text{v}} \|v\|$. \qed

**Corollary 4.14** (Careful CBV Correctness Criterion). If $\exists t. e \vdash_{\text{vc}} \|t\| \implies e \vdash_{\text{v}} \|e\|$. Proof. By the definition of $(\vdash_{\text{vc}})$, there is a sequence $e \rightarrow^*_{\text{vc}} e_1 \leftarrow^*_{\text{vc}} e_2 \rightarrow^*_{\text{vc}} \cdots e_k \rightarrow^*_{\text{vc}} \|t\|$. By the Careful Erasure Theorem, erasure converts all of these careful reductions to $(\vdash_{\text{v}})$. Therefore, noting that $(\vdash_{\text{vc}}) \subseteq (\vdash_{\text{v}})$ and that $\|\cdot\|$ is idempotent, $\|e\| \vdash_{\text{v}} \|\|t\|| = \|t\| \vdash_{\text{v}} e$. \qed

With this criterion, one can reason about CBV staged programs without leaving the CBV theory. However, as I will show in Section 4.3, the General Correctness Criterion can be as convenient as the Careful Correctness Criterion in some cases.
4.2 Example: Verifying Staged Gibonacci in Call-By-Name

Let me demonstrate the use of the General Correctness Criterion on \texttt{gibst} from Section 2.2. I will prove that this MetaOCaml function correctly implements the Gibonacci sequence in CBN, which is made quite easy with the General Correctness Criterion. Before presenting the proof, however, I need to specify how MetaOCaml constructs are interpreted in \( \lambda V \). All constructions shown below are standard; see [23, 14] for a thorougher general discussion of encodings.

\textbf{Definition 4.15 (Interpretation of MetaOCaml).} Constructs that have obvious counterparts in \( \lambda V \), namely \texttt{fun} \( x \to e \), variables, application, and staging constructs, denote their counterparts. A \textit{top-level} binding \texttt{let} \( x = e \) is interpreted as defining a metatheoretical abbreviation with evaluation, i.e. \( x \) is defined to be syntactically equal to \( v \) where \( e \Downarrow^0 v \). An \textit{internal} (non-top-level) binding \texttt{let} \( x = e \text{ in } t \) is interpreted as \( (\lambda x.t) e \). A top-level recursive binding \texttt{let rec} \( f \ x = e \) is formalized using a fixed-point operator as \texttt{let} \( f = \lambda x.\text{fix} (\texttt{fun} f \ x \to e) \ x \), where \( \text{fix} \) is a closed level-0 \( \lambda V \) term that obeys the law

\[
\text{fix} v^0 \xrightarrow{+}^0 v^0 (\lambda x.\text{fix} v^0 x) \text{ where } x \notin FV(v^0).
\]

Numbers, arithmetic, and conditionals are assumed to be Church-encoded.

\textbf{Remark 4.16.} Strictly speaking, a top-level binding behaves like an application, much like an internal binding. But such an interpretation would require the statements and proofs of correctness to be littered with remarks about when those bindings should be substituted for, which obfuscate the argument without adding any significant insights. I instead interpret top-level bindings as abbreviations, which is essentially the same as assuming that they are always substituted for beforehand. My interpretation accurately reflects MetaOCaml’s behavior as long as no top-level
binding has a divergent right-hand side.

**Notation.** Let \( \lceil k \rceil \) denote the level-0 value that encodes an integer \( k \). By abuse of notation, let the encoding of arithmetic operations be written simply as themselves, e.g. addition is just written as +.

The exact statement of correctness needs some care. The goal is to prove that \( \text{gibst} \) correctly implements the Gibonacci sequence, but the point of Erasure is not to try to prove this fact directly. Instead, we note that the erasure (or unstaged counterpart) \( \text{gib} \) is correct by construction, so equating \( \text{gibst} \) and \( \text{gib} \) is sufficient. The virtue of a high-level, clearly structured program shows up here. Such a program is close to the mathematical or informal specification so that its correctness is easy to verify. Unfortunately, \( \text{gibst} \) and \( \text{gib} \) are not completely equivalent because \( \text{gibst} (-1) \) diverges immediately whereas \( \text{gib} (-1) \) gets partially applied and waits for the two remaining parameters before diverging. This kind of discrepancy between the staged and unstaged programs is quite common. Practically, it is easy to circumvent but is nonetheless a pitfall that the programmer must keep an eye on.

A formal proof can get around this problem in two ways. One is to restrict the functions’ (intended) domains. In the case of \( \text{gibst} \), the programmer probably does not intend this function to be applied to a negative argument, so we could say that verifying the behavior at non-negative arguments is sufficient. The other approach, which I demonstrate here, is to fully apply the functions before comparison.

**Lemma 4.17.** \( \text{gib} \equiv_n \|\text{gibst}\| \).

**Proof.** By inspecting Figure 2.1, one can readily see \( \|\text{gibgen}\|=\text{gib} \). Then
\[
\|\text{gibst}\| = (\text{fun} \ n \ x \ y \to \|\text{gibgen}\| n .<x> .<y>.) \equiv_n \text{gib}
\]
where the last equality is derived by $\beta$-substituting $n$, .\$x$. , and .\$y$. into $\text{gibgen}$.

\[\text{Corollary 4.18 (Correctness of gibst in CBN).} \text{ For all integers } k, k_x, \text{ and } k_y,\]

\[\text{gibst} \downarrow \gamma k \gamma \downarrow \gamma k_x \gamma \downarrow \gamma k_y \gamma \approx_n \text{gib} \downarrow \gamma k \gamma \downarrow \gamma k_x \gamma \downarrow \gamma k_y \gamma. \quad (\ast 11)\]

\[\text{Proof.} \text{ For } k < 0, \text{ both sides of the equation are closed and divergent at level 0; all closed divergent terms at the same level are equivalent (see Example 5.6 for a formal proof)}, \text{ so (\ast 11) holds for } k < 0. \text{ For } k \geq 0, \text{ it is sufficient (and possible) to prove } \text{gibst} \downarrow \gamma k \gamma \Downarrow \gamma n \text{gib} \downarrow \gamma k \gamma. \text{ Observe that the correctness criterion, Corollary 4.9, reduces the problem to showing that } \text{gibst} \downarrow \gamma k \gamma \text{ is provably equal to an unstaged term. By } \beta\text{-substitution},\]

\[\text{gibst} \downarrow \gamma k \gamma \Downarrow \gamma n .! .\text{\fun x y } \rightarrow \text{.\~}(\text{gibgen} \downarrow \gamma k \gamma . \text{\<x>. .\<y>.})>.\]

The right-hand side equals an unstaged term if the call to $\text{gibgen}$ yields a term of the form .\$\|e\|\$. , which is easily shown by induction on the number $k$ (not to be confused with the encoding $\downarrow \gamma k \gamma$). This condition can be checked easily with successive reductions.

\[\text{[If } k = 0 \text{]} \text{ Clearly } \text{gibgen} \downarrow \gamma 0 \gamma . \text{\<x>. .\<y>. } \rightarrow^*_n . \text{\<x>..}\]

\[\text{[If } k = 1 \text{]} \text{ Clearly } \text{gibgen} \downarrow \gamma 1 \gamma . \text{\<x>. .\<y>. } \rightarrow^*_n . \text{\<y>..}\]

\[\text{[If } k > 2 \text{]} \text{ The inductive hypothesis is that } \text{gibgen} \downarrow \gamma k - i \gamma . \text{\<x>. .\<y>. reduces to some .\|$\|e_{k-i}\|$$\$. for } i = 1, 2. \text{ Straightforward reduction then shows}\]

\[\text{gibgen} \downarrow \gamma k \gamma . \text{\<x>. .\<y>. } \rightarrow^*_n . \text{\|$\|e_{k-i}\|$$ + \|$\|e_{k-2}\|$$>.}\]

whose right-hand side is of the form .\$\|e\|\$. , as required.

Thus $\text{gibgen} \downarrow \gamma k \gamma . \text{\<x>. .\<y>.}$ is indeed provably equal to .\$\|e\|\$. , hence is equal to the erasure by the General Correctness Criterion, whenever $k \geq 0$. Then by Lemma 4.17, $\text{gibgen} \ k \Downarrow \gamma n \text{gib} \ k$.\]
We can expect this line of reasoning to work for many programs, particularly for PE-like applications. Corollary 4.9 reduces the problem to showing that the code generator (\texttt{gibgen} in this example) terminates to some $\|e\|$, which is straightforward if the programmer designs the generator with such an invariant in mind. Adhering to this invariant is easy, since there is no point in generating a code value whose body contains brackets unless the program requires more than two stages, which is rare. This result then suggests a simple development methodology that helps with correctness, namely, to build up the code generator from components that generate code values without superfluous brackets.

4.3 Example: Verifying Staged Gibonacci in Call-by-Value

Due to the mismatch in evaluation strategies in the premise and conclusion, applying the General Correctness Criterion in CBV requires more care than in CBN. In this section I present two correctness proofs for the staged Gibonacci function in CBV. One uses the General Correctness Criterion and the other uses the Careful Erasure Theorem. (The latter can just as well use the Careful Correctness Criterion, but the Erasure Theorem better illuminates the proof technique.) Each of these examples illustrates a separate proof technique that mitigates the inconvenience in the CBV Erasure Theorem. Interpretations of MetaOCaml constructs are as explained in the previous section.

The first idea is to show that both the staged program and its erasure CBV-terminate to CBN-normal forms, i.e. expressions that do not reduce any further in CBN. As the normal forms must be CBN-equal by the General Criterion, they must be identical due to the Church-Rosser property. For Gibonacci, this proof appears to require integers to be included as constants and not as encodings in $\lambda^V$. The
reason is that although canonical Church-encoded numbers are CBN-normal forms, the returned values may be non-normal forms that happen to be CBN observationally equivalent to the canonical encoding. To rule out this possibility, we need to assume that integer encodings are unique.

**Definition 4.19.** Integers are uniquely encoded iff \( \doteq n \) implies \( k = n \).

Unique encoding is satisfied if, for example, integers are added as constants instead of encodings. The following proof assumes that my results have been extended to \( \lambda V \) with constants. I expect this extension to be a straightforward exercise, as with most other \( \lambda \) calculi. The precaution about \( \text{gibst} \) diverging earlier than \( \text{gib} \), discussed in the previous section, applies here as well.

**Lemma 4.20.** \( \|\text{gibst}\| \approx V \text{gib} \).

**Proof.** As in the CBN case (Lemma 4.17), one can readily see \( \|\text{gibgen}\| = \text{gib} \) and

\[
\|\text{gibst}\| = (\text{fun } n x y \rightarrow \|\text{gibgen}\| n x y) \approx V \text{gib}
\]

The only issue here is that substituting the variables \( n, x, \) and \( y \) into \( \text{gibgen} \) does not preserve CBV provable equality. However, the substitution preserves observational equivalence because of the EQ-VAR\( C/\beta \) rule. \( \square \)

**Notation.** Let \( \sigma \) range over capture-avoiding substitutions with finite support.

**Corollary 4.21** (Correctness of \( \text{gibst} \) in CBV Using the General Criterion). Assume that integers are uniquely encoded. Then for all integers \( k, k_x, \) and \( k_y, \)

\[
\text{gibst} \doteq k \doteq k_x \doteq k_y \approx V \text{gib} \doteq k \doteq k_x \doteq k_y.
\]

**Proof.** The first half of the proof is a verbatim copy of the CBN proof. For negative \( k \), both sides of the equation are closed and non-terminating, so they are
equivalent. For non-negative \( k \), by induction on \( k \) and the CBV part of the General Correctness Criterion (Corollary 4.9), 
\[ \text{gibst} \; \langle k \rangle \rightarrow^{*} \; (\text{fun} \; x \; y \rightarrow \| e \|) \quad \text{and} \quad \| \text{gibst} \| \; \langle k \rangle \rightarrow^{*} \; (\text{fun} \; x \; y \rightarrow \| t \|) \]
for some unstaged terms \( \| e \| \) and \( \| t \| \) such that \( \| e \| \triangleleft_{n} \| t \| \). The question then is whether \( \sigma\| e \| \triangleleft_{v} \sigma\| t \| \), under the substitution \( \sigma \equiv [\langle k \rangle x, \langle k \rangle y]/x, y \). So far we have
\[ \text{gibst} \; \langle k \rangle \langle k \rangle x \langle k \rangle y \triangleleft_{v} \sigma\| e \| \triangleleft_{n} \sigma\| t \| \triangleleft_{v} \| \text{gibst} \| \; \langle k \rangle \langle k \rangle x \langle k \rangle y . \]

Now, one can easily check that \( \exists n_e. \; \sigma\| e \| \Downarrow^{0}_{v} \; \langle n_e \rangle \)
because the generated code \( \| e \| \) only uses + which is a total operation from pairs of integers to integers. It is also readily checked that \( \exists n_t. \; \sigma\| t \| \Downarrow^{0}_{v} \; \langle n_t \rangle \). Then
\[ n_t \triangleleft_{v} \sigma t \triangleleft_{n} \sigma e \triangleleft_{v} n_e , \]
so \( n_e \triangleleft_{n} n_t , \) which by the unique encoding assumption implies \( n_e = n_t \). Therefore,
\[ \text{gibst} \; \langle k \rangle \langle k \rangle x \langle k \rangle y \triangleright_{v} \sigma \| e \| \triangleright_{n} n_e = n_t \triangleright_{v} \| \text{gibst} \| \; \langle k \rangle \langle k \rangle x \langle k \rangle y . \]
Then by Lemma 4.20,
\[ \text{gibst} \; \langle k \rangle \langle k \rangle x \langle k \rangle y \triangleright_{v} \| \text{gibst} \| \; \langle k \rangle \langle k \rangle x \langle k \rangle y \approx_{v} \text{gib} \; \langle k \rangle \langle k \rangle x \langle k \rangle y. \]

The second technique is speculative substitution, where values are substituted for variables without a priori justification. Then after showing \( \| \sigma e \| \triangleright_{v} \sigma e \) using careful reduction, where \( \sigma \) is the substitution of values for variables, the substitution is justified by retracing the careful reduction backwards. This strategy does not need to assume unique encoding of integers.

Lemma 4.22. \( \| \text{gibgen} \| = \text{gib} . \)

Proof. Immediate from definition.
Corollary 4.23 (Correctness of \texttt{gibst} in CBV Using Careful Erasure). For all integers \( \lceil k \rceil, \lceil k_x \rceil, \) and \( \lceil k_y \rceil, \)

\[
gibst \lceil k \rceil \lceil k_x \rceil \lceil k_y \rceil \approx_v \gib \lceil k \rceil \lceil k_x \rceil \lceil k_y \rceil.
\]

Unique encoding of integers is \textit{not} required.

\textit{Proof.} The argument for \( k < 0 \) is the same as before. For \( k \geq 0 \), by induction on \( k \) again, \( \gibgen \lceil k \rceil \cdot \langle x \rangle \cdot \langle y \rangle \cdot \) terminates to some code value of the form \( \langle \| e \| \rangle \).

by ordinary, non-careful reduction. Let the reduction trace be

\[
\gibgen \lceil k \rceil \cdot \langle x \rangle \cdot \langle y \rangle \cdot \rightarrow_v \sigma e_1 \rightarrow_v \sigma e_2 \cdots \rightarrow_v \sigma e_m \rightarrow_v \langle \sigma \| e \| \rangle \cdot \quad (\ast 12)
\]

This reduction is not careful because it \( \beta \) reduces recursive calls of the form

\[
\gibgen \lceil k \rceil \cdot \langle x \rangle \cdot \langle y \rangle \cdot
\]

The second and third arguments’ erasures are divergent, so the function call cannot be expanded by careful reduction. However, once values are substituted for the variables, this \( \beta \) reduction becomes careful; no other non-careful reduction step is required. Therefore, speculatively applying the substitution \( \sigma \equiv [\lceil k_x \rceil, \lceil k_y \rceil / x, y] \),

\[
\gibgen \lceil k \rceil \cdot \langle \sigma k_x \rangle \cdot \langle \sigma k_y \rangle \cdot \rightarrow_{\text{ve}} \sigma e_1 \rightarrow_{\text{ve}} \sigma e_2 \cdots \rightarrow_{\text{ve}} \sigma e_m \rightarrow_{\text{ve}} \langle \sigma \| e \| \rangle \cdot \quad (\ast 12)
\]

Then by the Careful Erasure Theorem,

\[
\| \gibgen \| \lceil k \rceil \lceil k_x \rceil \lceil k_y \rceil \rightarrow_v^* \sigma \| e \|.
\]

Now, to justify the speculative substitution, we factor out the \( \sigma \) as an application by reverse \( \beta \) reduction and rewind the reduction trace \((\ast 12)\).

\[
\sigma \| e \| \longleftarrow_v^* \cdot \langle \text{fun} \ x \ y \rightarrow \| e \| \rangle \cdot \lceil k_x \rceil \lceil k_y \rceil
\]

\[
\longleftarrow_v^* \cdot \langle \text{fun} \ x \ y \rightarrow \cdot \langle \gibgen \lceil k \rceil \cdot \langle x \rangle \cdot \langle y \rangle \rangle \cdot \lceil k_x \rceil \lceil k_y \rceil
\]

\[
\longleftarrow_v \gibst \lceil k \rceil \lceil k_x \rceil \lceil k_y \rceil.
\]
Putting the two results together, we have
\[ \| \text{gibgen} \| \xrightarrow{\sigma} \| e \| \xleftarrow{\text{gibst}} \]
\[ k \upharpoonright k_x \upharpoonright k_y \rightarrow^* v \]
\[ k \upharpoonright k_x \upharpoonright k_y \].

(13)

By Lemma 4.22, it follows that
\[ \text{gib} \xrightarrow{\sigma} \| \text{gibgen} \| \xleftarrow{\text{gibst}} \]
\[ k \upharpoonright k_x \upharpoonright k_y \approx_v \]
\[ k \upharpoonright k_x \upharpoonright k_y \].

At the heart of this proof is (13), which shows that a reduction under the influence of \( \sigma \) meets a reduction that leaves the arguments \( k_x \) and \( k_y \) on the side. This step is the reason that we had to trace the reduction of \( \text{gibgen} \) twice, with one time in reverse. The reduction on the left of (13) is necessary for bringing \( \text{gib} \) into the picture, and this reduction can be derived only under the influence of \( \sigma \). As the purpose of the reduction on the right of (13) is to relate the result \( \sigma \| e \| \) to \( \text{gibst} \), the reduction is naturally a reversal of the reduction on the left, except with the substitution \( \sigma \) factored out to the context.

As I mentioned above, this proof can use the Careful Correctness Criterion instead of the Careful Erasure Theorem. The proof would be slightly more concise, but with the downside that the intuitive motivation behind (13) would not be as clearly presented. Once the technique is understood, however, using the Careful Correctness Criterion with provable equality is in general more flexible and convenient than using the Careful Erasure Theorem with reduction.

The two proof techniques for CBV presented here have different trade-offs. Proving that \( e \) and \( \| e \| \) CBV-terminate to CBN-unique encodings works well for first-order functions. If first-order values are uniquely encoded (which is almost always the case in practice) and the program is type-checked, then the return type is sufficient to guarantee that return values, if any, are uniquely encoded. For example, MetaOCaml reports that the return type for \( \text{gibst} \) is an integer, which is a primitive built-in and
thus uniquely encoded. Then only CBV termination must be proved. The primary weakness of this technique is that it fails for higher-order return values, as higher-order values’ encodings are generally not unique.

A higher-order function such as an operation on lazy streams will probably benefit more from careful reduction. The primary strength is that the reasoning can be conducted entirely in CBV. The downside is that the prover must pay attention to the forms of reductions and rule out non-careful ones. Note, however, that with additional proof obligations this approach can handle non-careful \( \beta \) reductions like \((\lambda x. e^0) \langle \| \Omega \| \rangle \rightarrow^\nu [\langle \| \Omega \| \rangle / x] e^0\). If \( \| e^0 \| \) is strict in \( x \) in CBV, then \((\lambda x. \| e^0 \|) \| \Omega \| \approx^\nu \[\| \Omega \| / x\] \| e^0 \| \) holds since all divergent terms are equivalent (see Example 5.6). Thus in general, non-careful reductions can be handled by bridging sequences of (erasures of) careful reductions with strictness proofs. The reasoning principle in Chapter 5 may be useful for that task.

With careful reduction, one should keep in mind that the handling of higher-order data constructors is determined solely by their strictness and not the evaluation strategy of the overall language. For example, let \( :: \) and \( [] \) be the standard strict list constructors as found in MetaOCaml and consider the inference

\[
(e :: [] \rightarrow t :: []) \implies (\| e \| :: [] \rightarrow \| t \| :: [])
\]

Clearly, this inference in CBV requires the reduction on the left to be careful because \( e \) can be \((\lambda_\psi v^0) \langle \Omega \rangle\). The reduction must still check \( \| e \| \downarrow^0 \) even if the overall language is CBN, such as in Haskell, if the constructor itself is strict. Conversely, the reduction need not be careful in CBV languages like MetaOCaml if the data constructor is lazy.
Chapter 5

Extensional Reasoning for $\lambda^V$

A great advantage of functional programs based on the $\lambda$ calculus is the presence of extensively studied equational reasoning techniques, and a part of the goal of this thesis is to demonstrate that staging is not too invasive to such reasoning. An important class of such reasoning techniques is extensional reasoning, most notably Abramsky’s applicative bisimulation [1]. Applicative bisimulation is a sound and complete, coinductive redefinition of ($\approx$). It is extensional, equating terms by comparing only their externally observable behavior. These properties are in contrast to the sound but incomplete, inductive nature of ($\doteq$). Provable equality is intensional, equating terms by looking inside the syntactic structure of the terms to pattern-match and rewrite subterms by the equational axioms.

Applicative bisimulation is a cornerstone in the analysis of higher-order structures such as higher-order functions and lazy lists [13], which are essential to advanced functional programming. It is a natural desire to reproduce this tool in $\lambda^V$. This chapter formulates an analogue of applicative bisimulation and proves that the coinductive, extensional proof principle derived from that analogue is sound and complete.

In a series of examples, I demonstrate that the extensional proof principle concisely proves a number of examples that are otherwise nontrivial. However, one of the
examples that I will consider, namely the extensionality rule for functions

\[ \forall a. (\lambda x.e^0) a \approx (\lambda x.t^0) a \]

\[ \lambda x.e^0 \approx \lambda x.t^0 \]

\[ [\omega] \]

is not completely satisfactory. Unlike in the plain \( \lambda \) calculus, this rule for \( \lambda^V \) requires equivalence at open-term arguments. This principle is thus weaker than the statement that is typically accepted in the plain \( \lambda \) calculus, which only requires equivalence at closed arguments [28, 16]. In CBV, the weak \( \omega \) rule shown above is still useful, for example for justifying EQ-VARC\( \beta \); in CBN, however, this rule is trivial because the argument can be just a variable.

Whether the stronger version of \( \omega \) that only considers closed arguments holds in \( \lambda^V \) is an interesting question, the answer to which is unclear. On the one hand, having to require equivalence at open arguments is not surprising because functions can be actually applied to open terms during program execution (see Proposition 3.10). On the other hand, I was not able to find an example where open-term arguments make a difference, and it seems intuitively plausible that they never do. The gray status of the stronger \( \omega \) rule suggests that my version of the applicative bisimulation may still have room for improvement, and for this reason I feel that explaining the derivation of the proof principle in \( \lambda^V \) is as important as proving its legitimacy.

I will first present the coinductive extensional proof principle extracted from applicative bisimulation and demonstrate its use (Section 5.1). Coinduction is not necessary to understand the proof principle and its applications. Then after briefly reviewing coinduction (Section 5.2), I will explain in detail the derivation of applicative bisimulation in \( \lambda^V \) (Section 5.3). I prove that applicative bisimulation is a sound and complete reformulation of observational equivalence (Section 5.4). Then the soundness and completeness of the extensional reasoning principle follows.
5.1 Extensional Proof Principle

As the extensional proof principle is more complicated than the equational theory, it is sometimes useful to have a directed version of observational equivalence, called observational order, that only requires a half of what the full equivalence demands. With observational order, one can prove one half of observational equivalence at a time. Note that I am directing the open observational equivalence (Definition 3.9) rather than the closed one (Definition 3.8).

**Definition 5.1** (Observational Order). Define observational order \((\preceq)\) by directing the definition of observational equivalence:

\[
e \preceq t \iff \forall C. (C[e], C[t] \in E^0 \implies (C[e] \Downarrow^0 \iff C[t] \Downarrow^0)).
\]

**Proposition 5.2.** \((\preceq) \cap (\simeq) = (\approx)).

**Proof.** Obvious from definition.

The extensional proof principle to be verified in this chapter is formulated using observational order as follows. To show that a relation \(R\) implies \((\preceq)\), one proves \(R\) to be increasing. A relation \(R\) is increasing when \(eRt\) implies that \(t\) is more terminating than \(e\) under all substitutions \(\sigma\). Furthermore, whenever \(\sigma e\) and \(\sigma t\) terminate, the results of performing an experiment on their return values must be related by \(R\) or \((\simeq)\). An experiment on level-0 abstractions substitutes a common argument into the bodies, an experiment on level-0 code values applies \texttt{run} and eliminates the outermost brackets, and an experiment on any higher-level return values reinterprets them as lower-level expressions. Note the experiment on high-level return values is effectively a no-op. These definitions are formalized below.
Notation. A signature $\sigma : \text{Var} \to S$ specifies that $\sigma$ draws substitutes from the set $S$, that is $\sigma = [e_i/x_i]$ where $\forall i. e_i \in S$. By default, $\sigma$ has the signature $\sigma : \text{Var} \to A$ unless otherwise specified.

Definition 5.3. For a relation $R \subseteq E \times E$, set $e \sqsubseteq^\ell_R t$ iff $e, t \in E^\ell$ implies $\exists v. e \Downarrow^\ell v \implies \exists u. t \Downarrow^\ell u$ and, if $v, u$ exist, they satisfy:

- If $\ell = 0$ and $v = \lambda x.e'$, $u = \lambda x.t'$ and $\forall a. [a/x]e'R[a/x]t' \lor [a/x]e' \preceq [a/x]t'$.
- If $\ell = 0$ and $v = \langle e' \rangle$, $u = \langle t' \rangle$ and $e'Rt' \lor e' \preceq t'$.
- If $\ell > 0$, $vRu \lor v \preceq u$.

Then define $(\sqsubseteq^\ell_R)$ by the same conditions but using $(\iff)$ and $(\approx)$ instead of $(\implies)$ and $(\preceq)$, respectively. Say that $R$ is increasing iff $eRt \implies \forall \sigma : \text{Var}^\text{fin} \to A. \forall \ell. \sigma e \sqsubseteq^\ell_R \sigma t$, decreasing iff $eRt \implies \forall \sigma : \text{Var}^\text{fin} \to A. \forall \ell. \sigma e \sqsupseteq^\ell_R \sigma t$, and justified iff $eRt \implies \forall \sigma : \text{Var}^\text{fin} \to A. \forall \ell. \sigma e \sqapprox^\ell_R \sigma t$. Note that the $\sigma$ do not necessarily close the expressions to which they are applied, and that not all free variables are eliminated.

Definition 5.4. A relation $R$ is preserved by substitution iff $eRt \implies \forall \sigma : \text{Var}^\text{fin} \to A. (\sigma e)R(\sigma t)$.

Lemma 5.5 (Extensional Proof Principle). If a relation $R \subseteq E \times E$ is preserved by substitution, then

(i) $R$ is increasing iff $R \subseteq (\preceq)$.

(ii) $R$ is decreasing iff $R \subseteq (\preceq)$.

(iii) $R$ is justified iff it is increasing and decreasing.

(iv) $R$ is justified iff $R \subseteq (\approx)$.

Notice how this proof principle exploits well-leveled bindings. When $(\sqsubseteq^\ell_R)$ compares level-0 abstractions $\lambda x.e$ and $\lambda x.t$, it does not directly compare the bodies $e$
and \( t \) but substitutes away \( x \) beforehand, comparing \([a/x]e\) and \([a/x]t\) instead. In CBN this substitution has no significance because \( a \) ranges over \( A_n \) which contains \( x \), but in CBV the \( a \) is much better behaved than \( x \). For example, the application \((\lambda_x.d^0) \ a\) can be contracted but \((\lambda_x.d^0) \ x\) cannot in CBV. The substitution encodes the intuition that well-leveled variables can be treated as substitutable arguments, whereas treating potentially ill-leveled variables in that way is unsound.

Proving Lemma 5.5 is the goal of this chapter. Before I delve into the proof, however, I will demonstrate with a series of examples how this extensional reasoning principle can be used to derive equivalences which are impossible to prove directly with provable equality.

The first example shows that divergent terms are equivalent. Divergence is a purely extensional property, so this equivalence is impossible to derive in the intensional equational theory. The divergence must not be caused by getting stuck on a free variable, because in that case substitution may restore termination. More precisely, equivalence holds among terms that are at the same level, diverge at the same level, and whose divergence persists under substitutions. The last condition holds if the divergence is due to non-termination, term constructor mismatches like \( \langle e \rangle_t \) and \( \neg(\lambda x.e) \), or getting stuck on an ill-leveled variable bound within the divergent term.

**Example 5.6** (Equivalence of Divergent Terms). Let \( \Omega, \Omega' \in E^m \setminus E^{m-1} \) where \( E^{-1} \overset{\text{def}}{=} \emptyset \) when \( m = 0 \). If \( \sigma \Omega \uparrow^m \) and \( \sigma \Omega' \uparrow^m \) for all \( \sigma \), then \( \Omega \approx \Omega' \).

*Proof.* Set \( eRt \overset{\text{def}}{\iff} e, t \in E^m \setminus E^{m-1} \land \forall \sigma. (\sigma e \uparrow^m \land \sigma t \uparrow^m) \). Then \( \Omega R \Omega' \), so by Lemma 5.5, it suffices to show that \( R \) is justified. Furthermore, \( R \) is symmetric so proving it to be increasing is sufficient. Let \( (e, t) \in R \) be given, and fix \( \sigma \) and \( \ell \).

[If \( \ell < m \)] By Lemma 3.11 \( \sigma e, \sigma t \not\in E^\ell \), so \( \sigma e \leq_R \sigma t \) vacuously.
[If $\ell = m$] By assumption $\sigma e \mathrel{\uparrow}^\ell$, so $\sigma e \subseteq_R^\ell \sigma t$ vacuously.

[If $\ell > m$] By Lemma 3.11 $\sigma e, \sigma t \in V^\ell$, so these expressions terminate to themselves. Then $(\sigma e)R(\sigma t)$ by definition, so $\sigma e \subseteq_R^\ell \sigma t$. \hfill \Box

That this equivalence is undervariable in the equational theory formally follows from the Church-Rosser property. Similar results can be proved for subsequent examples (except for the $E_V$ example immediately below), which I omit.

**Proposition 5.7.** Let $\Omega \overset{\text{def}}{=} (\lambda x.x) (\lambda x.x)$. Then $\Omega \neq \Omega \lambda y.y$ although $\Omega \approx \Omega \lambda y.y$ as shown above.

**Proof.** The only redex in $\Omega$ and $\Omega \lambda y.y$ is $\Omega$, and $\Omega$ only reduces to itself. Therefore, $\Omega$ and $\Omega \lambda y.y$ only reduce to themselves, so these expressions never meet at a common reduct. Hence by the Church-Rosser property, they cannot be provably equal. \hfill \Box

The ability to derive strictly more equivalences than the intensional equational theory makes the extensional proof principle useful for expanding the equational theory. The primordial example is $RED-E_V$: I first verified the soundness of this rule by the extensional proof principle rather than Takahashi’s method presented in Chapter 3.

**Example 5.8** (Soundness of $E_V$). $\tilde{\langle} e \tilde{\rangle} \approx e$.

**Proof.** Set $R \overset{\text{def}}{=} \{ (\tilde{\langle} t \rangle, t) : t \in E \}$. Take an arbitrary pair $(\tilde{\langle} t \rangle, t) \in R$ and fix $\sigma, \ell$.

[If $\tilde{\langle} t \rangle, t \in E^\ell$] $\tilde{\langle} t \rangle$ has level at least 1, so $\ell > 0$. To see that $\sigma e \parallel_R^\ell \sigma t$:

[If $\ell = 1$] $\sigma^- (t) \Downarrow^\ell v \iff \sigma t \Downarrow^\ell v$. Then clearly $v \approx v$.

[If $\ell > 1$] $\sigma^- (t) \Downarrow^\ell \tilde{\langle} v \tilde{\rangle} \iff \sigma t \Downarrow^\ell v$, and $\tilde{\langle} v \tilde{\rangle} R v$ by definition.

[If $\tilde{\langle} t \rangle, t \notin E^\ell$] By Lemma 3.11 $\sigma^- (t), \sigma t \notin E^\ell$, so $\sigma e \parallel_R^\ell \sigma t$ vacuously. \hfill \Box

Therefore, $R$ is justified. By Lemma 5.5, $\tilde{\langle} e \tilde{\rangle} \approx e$ follows.
I expect more rules to exist that extensional reasoning can justify; the following rule is an example. Every such rule is a valid extension to the equational theory, but other factors affect whether the rule can mix in well with the rest of the theory. In particular, adding a rule that compromises confluence and/or the Erasure Theorem is probably not a good extension to the theory. I excluded the rule in the following example from the equational theory in this thesis partly because I could not check its impact on confluence and erasure due to lack of time.

Example 5.9. \(\text{run } \langle \neg e \rangle \approx \text{run } e\).

Proof. Set \(R \overset{\text{def}}{=} \{(\text{run } \langle \neg t \rangle, \text{run } t) : t \in E\}\). Take an arbitrary pair \((\text{run } \langle \neg t \rangle, \text{run } t) \in R\) and fix \(\sigma, \ell\). I will show \(\text{run } \langle \neg t \rangle \Downarrow_{R}^{\ell} \text{run } t\). Assume \(\sigma(\text{run } \langle \neg t \rangle), \sigma(\text{run } t) \in E^{\ell}\). Cases are split on \(\ell\).

[If \(\ell = 0\)] If \(\sigma(\text{run } \langle \neg t \rangle) \Downarrow^{0} v\), then

\[
\begin{align*}
(i) & \quad \sigma t \Downarrow^{\ell} \langle d \rangle \quad (ii) \quad d \Downarrow^{0} v \\
& \quad \text{by inversion}
\end{align*}
\]

\(\sigma(\text{run } t) \rightsquigarrow_{0}^{\ast} \text{run } \langle d \rangle \rightsquigarrow_{0} \Downarrow^{0} v \quad \text{immediately}\)

Conversely, if \(\sigma(\text{run } t) \Downarrow^{0} v\), then

\[
\begin{align*}
(i) & \quad \sigma t \Downarrow^{0} \langle d \rangle \in V^{0} \quad (ii) \quad d \Downarrow^{0} v \\
& \quad \text{by inversion}
\end{align*}
\]

\(\sigma(\text{run } \langle \neg t \rangle) \rightsquigarrow_{0}^{\ast} \text{run } \langle d \rangle \rightsquigarrow_{0} \Downarrow^{0} v \quad \text{immediately}\)

Thus \(\sigma \text{run } \langle \neg t \rangle \Downarrow^{0} v \iff \sigma \text{run } t \Downarrow^{0} v\) and \(v \approx v\) by reflexivity.

[If \(\ell > 0\)] If \(\sigma(\text{run } \langle \neg t \rangle) \Downarrow^{\ell} v\), then

\[
\begin{align*}
(i) & \quad \sigma t \Downarrow^{\ell} v' \in V^{\ell} \quad (ii) \quad v = \text{run } \langle \neg v' \rangle \\
& \quad \text{by inversion}
\end{align*}
\]

\(\sigma(\text{run } t) \Downarrow^{\ell} \text{run } v' \in V^{\ell} \quad \text{immediately}\)
Conversely, if $\sigma(\text{run } t) \Downarrow^0 v$, then

$$(iii) \quad \sigma t \Downarrow^0 v' \in V^\ell \quad (iv) \quad v = \text{run } v'$$

by inversion

$$\sigma(\text{run } \langle \tilde{\ell} \rangle) \sim_{0}^* \text{run } \langle \tilde{\ell} v' \rangle \in V^\ell$$

immediately

Thus $\sigma(\text{run } \langle \tilde{\ell} \rangle) \Downarrow^\ell \text{run } \langle \tilde{\ell} v' \rangle \iff \sigma(\text{run } e) \Downarrow^\ell \text{run } v'$ and $(\text{run } \langle \tilde{\ell} v' \rangle) R(\text{run } v')$ by definition.

By Lemma 5.5, it follows that $R \subseteq (\sim)$, hence $\text{run } \langle \tilde{e} \rangle \approx \text{run } e$. 

The last example is the $\omega$ rule mentioned at the beginning of the chapter, which states that functions behaving equivalently on all arguments are themselves equivalent. This example is significant for two reasons. One reason was discussed above, namely that it suggests that the extensional proof principle may have room for improvement. The other reason is that this example explains why $(\subseteq_{R}^\ell)$ should compare the bodies of abstractions only under substitutions.

**Lemma 5.10.** $\forall a. [a/x] e \approx [a/x] t \implies \forall \sigma. [a/x] \sigma e \approx [a/x] \sigma t$.

**Proof.** The substitutions $[a/x]$ and $\sigma$ can be commuted without changing the result of applying the substitutions to $e$ and $t$, modulo provable equality; the conclusion is then immediate. See Section A.5 for details. 

**Example 5.11.** The $\omega$ rule is sound in $\lambda^V$, i.e. if $\lambda x. e, \lambda x. t \in E^0$ and

$$(*)_{14} \quad \forall a \in V^0. (\lambda x. e) \ a \approx (\lambda x. t) \ a,$$

then $\lambda x. e \approx \lambda x. t$.

**Proof.** Take $R \overset{\text{def}}{=} \{(\sigma \lambda x. e, \sigma \lambda x. t) \in E^0 \times E^0 : \text{all } \sigma\}$. Clearly, $R$ is preserved under substitution. For every $\sigma$, both $\sigma \lambda x. e$ and $\sigma \lambda x. t$ terminate to themselves at all levels because they are in $V^0$ by Lemma 3.11.
By Barendregt’s variable convention [3], \( \sigma \lambda x.e = \lambda x.\sigma e \) and \( \sigma \lambda x.t = \lambda x.\sigma t \). Then for any \( a \in A \), Lemma 5.10 and \((\star 14)\) guarantee \( [a/x]\sigma e \approx [a/x]\sigma t \).

Therefore \( \lambda x.e \not\equiv_R \lambda x.t \), and by Lemma 5.5, \( R \subseteq (\approx) \). In particular, \( \lambda x.e \approx \lambda x.t \).

Notice that the proof would get stuck in the level 0 case in CBV if \((\equiv_R)\) did not compare the bodies of \( \lambda x.\sigma e \) and \( \lambda x.\sigma t \) under the substitution \([a/x]\) but required \( \sigma e \approx \sigma t \) directly. In CBV, showing \( \sigma e \approx \sigma t \) is strictly more difficult because the information that \( x \) is well-leveled is lost. In particular, if \( e = (\lambda . v^0) \) and \( t = v^0 \), then \( e \not\approx t \) but \([a/x]e \approx [a/x]t \). Thus exploitation of well-leveled bindings is an essential feature of the proof principle. This feature will turn out to be a major source of complication for proving the soundness of the proof principle, but as this example shows the effort is worthwhile; a proof principle that does not exploit well-leveled bindings is incomplete in CBV. As noted before, exploitation of well-leveled bindings is a concept that only applies to CBV. In CBN, \((\star 14)\) directly implies \( e \approx t \) and hence \( \lambda x.e \approx \lambda x.t \) because \( a \) ranges over \( E^0 \), including \text{Var}.

With Example 5.11, I can make good on the promise from Subsection 3.3.2 to justify the EQ-VARC\( \beta \) rule.

**Proposition 5.12.** The following rule holds in \( \lambda^V \):

\[
\frac{C[(\lambda y.e^0) \ x] \in E^0 \quad C \text{ does not bind } x}{\lambda x.C[(\lambda y.e^0) \ x] \approx^* \lambda x.C[\text{[x/y]}e^0]} \quad \text{[EQ-VARC}\beta\text{]}
\]

**Proof.** If I replace \( x \) by an arbitrary argument \( a \), then \((\lambda y.e^0) \ a \ = \ [a/y]e^0 \), so \([a/x](C[(\lambda y.e^0) \ x]) \ = \ [a/x](C[\text{[x/y]}e^0]) \). Then the conclusion follows by the \( \omega \) rule. \( \square \)
5.2 A Coinduction Primer

Before explaining applicative bisimulation for $\lambda^V$ and proving Lemma 5.5, I will give a brief introduction to coinduction as this notion is not as standard as induction. A more thorough treatise is given in [12].

Coinduction is the dual to induction. A coinductive definition finds the greatest fixed point of a set of derivations, whereas an inductive definition finds the least fixed point. Coinduction on a coinductive set $S$ shows that a certain property implies membership in $S$, whereas induction on an inductive set $S'$ shows that membership in $S'$ implies a certain property. Construction of the fixed point relies on the Knaster-Tarski Fixed Point Theorem, from which the associated principle of coinduction falls out as a bi-product.

Definition 5.13. A complete lattice is a triple $(\mathcal{L}, \leq, \bigvee)$ such that $(\mathcal{L}, \leq)$ forms a partial order in which every subset $S \subseteq \mathcal{L}$ has a least upper bound $\bigvee S$ in $\mathcal{L}$. An upper bound for $S$ is an element $y \in \mathcal{L}$ such that $\forall x \in S. \ x \leq y$, and the least upper bound for $S$ is the least such element, i.e.

$$\forall y \in \mathcal{L}. \ (\forall x \in S. \ x \leq y) \implies \bigvee S \leq y.$$  

By abuse of terminology the set $\mathcal{L}$ by itself may also be called a complete lattice, with $(\leq)$ and $\bigvee$ to be inferred from context.

Remark 5.14. This definition forces the existence of greatest lower bounds, in accord with the standard definition of complete lattice. I will only be concerned with upper bounds and maximal fixed points, however.

Theorem 5.15 (Knaster-Tarski Fixed Point). Let $f : \mathcal{L} \to \mathcal{L}$ be a function from a complete lattice $\mathcal{L}$ to itself. If $f$ is monotone—$(x \leq y)$ implies $(f \ x \leq f \ y)$—then $f$ has a greatest fixed point $z$ which is also the greatest element such that $z \leq f \ z$. 

Proof. Take $S = \{ x \in L : x \leq f x \}$ and $z = \bigcup S$. Then

\begin{align*}
\forall x \in S. x & \leq z \quad \text{because } z \text{ is an upper bound} \\
\forall x \in S. f x & \leq f z \quad \text{by monotonicity} \\
z & \leq f z \quad \text{because } z \text{ is the least upper bound} \\
z & \in S \quad \text{by definition of } S \\
f z & \leq f (f z) \quad \text{by (1) and monotonicity} \\
f z & \in S \quad \text{by definition } S \\
z & \geq f z \quad \text{because } z \text{ is an upper bound} \\
z & = f z \quad \text{by (1)(2)}
\end{align*}

Clearly every fixed point of $f$ and every element $x \in L$ such that $x \leq f x$ are in $S$, so $z$ is the greatest of such elements. \hfill \square

The specific complete lattices I need are powerset lattices and product lattices. Both constructions are standard. I omit the straightforward proof that a powerset lattice is a complete lattice.

**Definition 5.16.** A powerset lattice of a set $S$ is the complete lattice $(\wp S, \subseteq, \bigcup)$ where $\wp S$ denotes the powerset of $S$.

**Definition 5.17.** If $(L_i, \leq_i, \bigcup_i)_{i \in I}$ is a family of complete lattices, then its product is the triple $(\prod_{i \in I} L_i, \leq, \bigcup)$ where the ordering operators are defined component-wise:

\begin{align*}
\prod_{i \in I} x_i & \leq \prod_{i \in I} y_i \iff \forall i \in I. \ x_i \leq_i y_i \\
\bigcup_{i \in I} S_i & \equiv \bigcup_i S_i^{i \in I}
\end{align*}

where $S_i$ is \{ $x_i : \prod_{j \in I} x_j \in S$ \}, the set of the $i$-th components of all sequences in $S$.

**Proposition 5.18.** A product of complete lattices is always a complete lattice.
Proof. The \((\leq)\) relation clearly inherits reflexivity and transitivity from \((\leq_i)\), so \(\prod_i \mathcal{L}_i\) is a partial order. For \(\bigcup\), let a subset \(S \subseteq \prod_i \mathcal{L}_i\) be given and set \(\overline{z}_i \overset{\text{def}}{=} \bigcup S\). For an arbitrary \(\overline{z}_i \in S\), by definition \(\forall i. \ x_i \leq z_i\) so \(\overline{z}_i \leq \overline{z}_i\). Therefore, \(\overline{z}_i\) bounds \(S\) in \(\prod_i \mathcal{L}_i\). For any upper bound \(\overline{y}_i\) of \(S\), for every \(i\), the \(y_i\) bounds \(S_i\) in \(\mathcal{L}_i\) so \(z_i \leq y_i\). Therefore, \(\overline{z}_i \leq \overline{y}_i\) so \(\overline{z}_i\) is the least upper bound of \(S\) in \(\prod_i \mathcal{L}_i\). \(\square\)

As no other notion of product will be used, the following notation will not be confusing.

**Notation.** If \(R\) is a binary relation, \(\overline{x}_i R \overline{y}_i\) means \(\forall i. \ x_i R y_i\).

A coinductive definition of a set \(S\) in a universe \(U\) is a self-referential definition of the form \(S \overset{\text{def}}{=} \max f S\) for some monotonic \(f : \wp U \rightarrow \wp U\), where \(S\) is taken to be the largest solution of the equation. If \(S\) is a binary relation, I may instead write \(xSy \overset{\text{def}}{=} \max \phi(S, x, y)\) for some predicate \(\phi\), which should be interpreted as \(S \overset{\text{def}}{=} \{(x, y) : \phi(S, x, y)\}\). The Knaster-Tarski Fixed Point Theorem guarantees the existence of the equation’s solution as well as the associated principle of coinduction:

\[
\forall T \subseteq \mathcal{L}. \ T \subseteq f \ T \implies T \subseteq S.
\]

Thus to show that some property \(\phi\) implies membership in \(S\), one only needs to show that for some \(T \supseteq \{x \in \mathcal{L} : \phi(x)\}\), it is the case that \(T \subseteq f \ T\).

To a first approximation, the purpose of this chapter is to find an \(f : \wp(E \times E) \rightarrow \wp(E \times E)\) such that \((\approx) = \max f(\approx)\). Then showing \(e \approx t\) reduces to finding a set \(S\) such that \((e, t) \in S\) and \(S \subseteq f S\), which can be significantly easier than considering the behaviors of \(e\) and \(t\) under arbitrary contexts. A coinductive redefinition of \((\approx)\) of the form \((\approx) = \max f(\approx)\), however, cannot seem to exploit well-leveled bindings. I solve this problem by finding a family of generalizations \((\approx_X) = \max_{X \in \wp \text{fin} \text{Var}} f(\approx_X)\) indexed by finite sets of variables \(X\) such that a mutually coinductive redefinition \((\approx_X) = \max f(\approx_X)\)
exists. This mutual coinduction is interpreted in the product \( \prod_{X \in \mathcal{P} \text{fin} \mathbf{Var}} \varphi(E \times E) \).

5.3 Deriving Applicative Bisimulation for \( \lambda^V \)

I will now derive my formulation of applicative bisimulation for \( \lambda^V \). As mentioned at the beginning of the chapter, I believe that the derivation is as important as the results. I will start with Howe’s formulation of applicative (bi)simulation [15] and motivate the modifications that are necessary to cope with the open-term semantics of \( \lambda^V \).

5.3.1 Howe’s Method

This subsection reviews Howe’s formulation of applicative (bi)simulation [15] and sketches his proof of soundness and completeness. The purpose of introducing his results are to illustrate the process of deriving the relation for \( \lambda^V \). As such, I will keep this presentation informal and only set off full-fledged formal definitions, lemmas, and proofs with headings if they transfer to \( \lambda^V \).

Howe formulates his proofs in a highly generic manner. I do not need this genericity because applying his method to \( \lambda^V \) requires generalization in a dimension that Howe did not consider; however, I will borrow the following notation as it helps to condense definitions and proofs. This notation automatically makes the presentation fairly generic.

**Notation.** Let \( \tau \) stand for a term constructor. In \( \lambda^V \), a term constructors is one of \( (\lambda x. \bullet) \), \( (\bullet \bullet) \), \( (\bullet) \), \( (\cdot) \), \( (\overline{\cdot}) \), and \( (\text{run} \bullet) \). Application of a term constructor to terms is written \( \tau \overline{e} \overline{f} \). Note that \( (\bullet \bullet) \) is a binary term constructor, where the two holes may be plugged with different expressions.
Howe’s method relies on the notion of closing substitutions, as do many proofs about the plain $\lambda$ calculus. Closing substitutions simplify proofs by eliminating free variables, but this trick will turn out to be a major problem in $\lambda V$.

**Notation.** Let $S_{cl}$ denote restriction of the set $S$ to closed terms, e.g. $V_{cl}^0$ is the set of all closed level-0 values.

**Definition 5.19 (Closing Substitution).** A substitution $\sigma$ *closes*, or is a *closing substitution* for, an expression $e$ iff $\sigma e \in E_{cl}$. A $\sigma$ closes a set of expressions iff the $\sigma$ closes all expressions in the set.

Howe’s definition of applicative bisimulation, which I write ($\sim^H$), is defined as the symmetric reduction of applicative simulation ($\preceq^H$). Informally $e \preceq^H t$ holds iff, coinductively,

- The term $t$ is at least as terminating as $e$ under all closing substitutions $\sigma$.
- The terms $\sigma e$ and $\sigma t$ terminate to values of the same form.
- These conditions propagate to immediate subterms of the values of $\sigma e$ and $\sigma t$.

Note that the last clause contains a self-reference. More formally,

$$eR^o t \iff \forall \text{closing } \sigma. \ (\sigma e)R(\sigma t) \quad \text{for relation } R \text{ on closed terms}$$

$$v \{R\}^H u \iff (v = \tau \overline{e_i} \implies (u = \tau \overline{t_i} \land \overline{\tau^R e_i})) \quad \text{for } v, u \text{ closed}$$

$$e[R]^H t \iff (e \downarrow v \implies (t \downarrow u \land v \{R\}^H u)) \quad \text{for } e, t \text{ closed}$$

$$\preceq^H \overset{\text{def}}{=} \max \{ \preceq^H \}$$

$$\sim^H \overset{\text{def}}{=} (\preceq^H) \cap (\preceq^H)$$

where $\tau$ ranges over the term constructors for the language under consideration.

The $R^o$ construction generalizes a closed-term relation $R$ to open terms via closing substitutions. Applicative simulation ($\preceq^H$) is defined as the greatest fixed point of
$R \mapsto [R]^H$. Then $[R]^H$ requires that the right-hand side is at least as terminating as the left-hand side, and asks $\{R\}^H$ to check that the values are of the same form and that their immediate subterms are related by $R$, where values have the same form if their outermost term constructors match. Then applicative bisimulation ($\sim^H$) is defined as the symmetric reduction of applicative simulation.

With some assumptions about the language under consideration (which are reasonable for single-stage calculi), Howe proves that ($\sim^{H_0}$) coincides with the observational equivalence of the language, which I will write as ($\approx^H$) to avoid confusion with $\lambda^V$'s equivalence. The central lemma is that ($\lesssim^{H_0}$) is context-respecting, which implies that ($\sim^{H_0}$) is a congruence.

**Definition 5.20.** A binary relation $R$ is context-respecting, or respects contexts, iff $\overline{e}_i R \overline{t}_i \implies (\tau \overline{e}_i) R (\tau \overline{t}_i)$. A context-respecting preorder is a precongruence, and a context-respecting equivalence relation is a congruence.

Once ($\sim^{H_0}$) is shown to be a congruence, the containment ($\sim^{H_0} \subseteq \approx^H$) is easy to prove: ($\sim^{H_0}$) directly implies equitermination, so by the context-respecting property it implies equitermination under all contexts. For the reverse containment, Howe proves ($\approx^H_{cl} \subseteq [\approx^H_{cl}]^H$), which is straightforward for the plain $\lambda$ calculus. Then ($\approx^H_{cl} \subseteq \sim^H$) by coinduction. With some work, this containment can be converted to ($\approx^H \subseteq \sim^H$).

Howe’s novelty lies in his proof of the context-respecting property of ($\lesssim^0$). The centerpiece of his elegant proof is the precongruence candidate ($\hat{\lesssim}^H$). He describes this relation as: “Informally, $e \hat{R}^H t$ if $t$ can be obtained from $e$ via one bottom-up pass of replacements of subterms by terms that are larger under $R$” [15]* where $t$ is

---

*The mathematical notation in this quote is changed to fit mine. It is otherwise a verbatim quote.
“larger” than \( e \) if \( eRt \). Derivation rules for Howe’s precongruence candidate are as follows.

\[
\begin{align*}
\frac{xR^c t}{x \hat{R} t} & \quad \text{[HPC-VAR]} \\
\frac{\overline{\tau} \overline{d}_i \overline{R} \overline{d}_i \overline{R}^c t}{(\tau \overline{d}_i) \hat{R} t} & \quad \text{[HPC-IND]}
\end{align*}
\]

I advise the reader to keep in mind which relations are for closed terms and which ones are for open terms in order to avoid confusion:

\[
(\preceq^H) \subseteq E_{cl} \times E_{cl} \quad [\preceq^H]^H \subseteq E_{cl} \times E_{cl} \quad \{\preceq^H\}^H \subseteq V_{cl} \times V_{cl}
\]

\[
(\preceq^{H^o}) \subseteq E \times E \quad (\preceq^{H^o}) \subseteq E \times E
\]

Induction shows that \((\preceq^{H^o})\) is reflexive, context-respecting, and \((\preceq^{H^o}) \subseteq (\preceq^{H})\). Howe proves \((\preceq^{H^o}_{cl}) \subseteq [\preceq^{H}]\), which by coinduction implies that \((\preceq^{H^o}_{cl}) \subseteq (\preceq^{H})\).

Then with a lemma showing that substitution preserves \((\preceq^{H})\), it follows that \((\preceq^{H}) \subseteq (\preceq^{H^o})^o \subseteq (\preceq^{H^o})^o\), so \((\preceq^{H}) = (\preceq^{H^o})\). Thus \((\preceq^{H^o})\) must be context-respecting as well.

The key property of the precongruence candidate that eases this series of proofs is closedness under substitution.

**Definition 5.21.** A relation \( R \) is **closed under substitution** iff \((eRt \land aRb) \Rightarrow ([a/x]e)R([b/x]t)\).

Note that closure under substitution is a stronger condition than being preserved by substitution, as the latter only considers applying the same substitution to \( e \) and \( t \). Closure under substitution has two important uses. One is to reason about application during the proof of \((\preceq^{H^o}_{cl}) \subseteq [\preceq^{H}]\). Consider a pair of expressions such that \( e_1 \preceq^{H} e_2 \) in the plain \( \lambda \) calculus where the next small-step is \( \beta \) reduction:

\[
e = (\lambda x.e') \ a \leadsto [a/x]e'

\text{where } a = \lambda y.a'

\]

\[
t = (\lambda x.t') \ b \leadsto [b/x]t'

\text{where } b = \lambda y.b'

\]
For simplicity, assume that \( e_1 \) and \( e_2 \) are closed. The goal \( e_1 \mathbin{\Rrightarrow}^H e_2 \) demands that we show \( e_2 \downarrow \) assuming \( e_1 \downarrow \). Some trial and error suggests that the only sensible strategy here is to induct on the number of steps that \( e_1 \) takes to terminate and to apply the inductive hypothesis to \([a/x]e'\). This reasoning requires proving \([a/x]e' \mathbin{\hat{\lesssim}}^H [b/x]t'\). But applying the inductive hypothesis to subterms \( \lambda x.e' \) and \( a \) of \( e_1 \) yields

\[
\begin{align*}
(i) \quad e' & \mathbin{\hat{\lesssim}}^H t' \\
(ii) \quad a' & \mathbin{\hat{\lesssim}}^H b'.
\end{align*}
\]  

(*16)

The context-respecting property is sufficient to convert (ii) to \( a \mathbin{\hat{\lesssim}} b \), but closure under substitutions is then required to establish \([a/x]e' \mathbin{\hat{\lesssim}}^H [b/x]t'\).

The same problem arises in \( \lambda^V \) for applications at level 0. This appears to be the hardest step of extensional reasoning. All of my earlier attempts at verifying extensional reasoning principles in \( \lambda^V \), including principles other than the one formalized as Definition 5.3 through Lemma 5.5, have failed essentially for this exact reason. Thus it is crucial to keep the precongruence candidate closed under substitution for \( \lambda^V \).

The other use of this property is that by reflexivity, closure under substitution ensures preservation by substitution. I stress that reflexivity is absolutely necessary here. As I will show shortly, the need to realize both reflexivity and closure under substitution forces applicative bisimulation to be an indexed family of relations instead of a single relation.

5.3.2 Adjusting to \( \lambda^V \)

Howe’s definitions need to be massaged before they can be interpreted in \( \lambda^V \). The most obvious problem is that \( \lambda^V \) does not have a unique notion of termination but defines termination differently for each level. This problem is easily solved by separately defining the relation for each level. Modifying the definition of \([R]^H\), I get the
tentative definition

\[ v \{ R \}^{I,0} u \overset{\text{def}}{\longleftarrow} (v = \tau \overline{e_i} \implies (u = \sigma \overline{t_i} \land \overline{e_i} R^o \overline{t_i})) \]

\[ v \{ R \}^{I,\ell+1} u \overset{\text{def}}{\longleftarrow} v R^o u \]

\[ e[R]^I t \overset{\text{def}}{=} (\forall \ell. (e, t \in E^\ell \land e \Downarrow^\ell v) \implies (t \Downarrow^\ell u \land v \{ R \}^{I,\ell} u)) \quad \text{for } e, t \text{ closed} \]

\[ (\preceq^I) \overset{\text{def}}{=} \max [\sim^I]^I \]

\[ (\sim^I) \overset{\text{def}}{=} (\preceq^I) \cap (\succeq^I). \]

The first line expands in $\lambda^V$ to:

\[ v \{ R \}^{I,0} u \overset{\text{def}}{=} \begin{cases} 
  v = \lambda x. e \implies (u = \lambda x. t \land e R^o t) & \text{and} \\
  v = \langle e \rangle \implies (u = \langle t \rangle \land e R^o t)
\end{cases} \]

For $\ell = 0$ this definition is the same as before. If $\ell > 0$, then $e[R]^I t$ states that after resolving all outstanding escapes in $e, t$ at level $\ell$, the resulting $v, u \in V^\ell = E^{\ell-1}$ must be related by $(\preceq^I)$. Then since $(\preceq^I) = [\preceq^I]^I$, the $v, u$ are compared again at all levels, including level $\ell - 1$. If $\ell$ is chosen too high, for example $\ell = 1$ when $e, t \in E^0$, then $e, t$ both terminate to themselves and the assertion degenerates to a tautology. If $\ell$ is too low, for example $\ell = 0$ when $e = \tilde{x}$, then the check $e, t \in E^\ell$ makes the assertion hold vacuously.

The other, more grave problem is the pervasive $R^o$ construction. Doing away with free variables by closing substitutions is a standard trick for the plain $\lambda$ calculus, but the trick does not work well in an open-term semantics. As the following failed attempts show, with the definition using $R^o$ the containment $(\preceq_{cl}^I) \subseteq [\preceq^I]^I$ could not be proved.

**Failed Lemma 5.22.** $e \preceq_{cl}^I t \implies e[\preceq^I] t$.

**Proof Attempt.** Fix an $\ell$, assume $e \sim^\ell_n v$, and try to prove $t \Downarrow^\ell u$ with $v \{ \preceq^I \}^{I,\ell} u$. Induction is possible on size(e), on n, or on some lexicographical ordering of these
parameters. Consider the case $e = \lambda x.e'$ and $\ell > 0$.

\begin{align*}
(i) & \quad e' \Downarrow^{\ell} \\
(ii) & \quad e' \approx^I d' \\
(iii) & \quad \lambda x.d' \approx^I t
\end{align*}

by inversion

A proof of $\lambda x.d' \Downarrow^{\ell}$ is in order, which is to say $d' \Downarrow^{\ell}$. The only hope of obtaining this assertion seems to be to use the inductive hypothesis on (iii), but no inductive measure seems to justify this step. Recall that the inductive hypothesis only applies to closed terms. $e'$ has $x$ free, so some closed argument $a$ must be substituted for $x$, but in general $\text{size}([a/x]e') \geq \text{size}(e)$. This observation rules out induction on $\text{size}(e)$ or $(\text{size}(e), n)$.

Induction on $n$ is also nonviable because $e'$ takes $n$ steps to terminate, and so does $[a/x]e'$. Informally, as $e'$ terminates with $x$ free, the $x$ never occupies the hole of an evaluation context. Therefore, the $x$ can be replaced with any level-0 expression without affecting the small-steps until termination. Lexicographical induction does not help since, again, $\text{size}([a/x]e') \geq \text{size}(e)$.

\[\square\]

A sensible workaround to try may be to prove the assertion for the open variants of the relations. But this approach also fails for the same reason as above.

**Failed Lemma 5.23.** $e \approx^I t \implies e'[\approx^I]^{\ell} t$.

**Proof Attempt.** Fix an $\ell$ and a closing $\sigma$, assume $\sigma e \rightsquigarrow^{\sigma} v$, and try to prove $\sigma t \Downarrow^{\ell} u$ with $v \{\approx^I\}^{\ell} u$. Induction is possible on the same measures as before. Consider the case $e = \lambda x.e'$ and $\ell > 0$.

\begin{align*}
\sigma(\lambda x.e') &= \lambda x.\sigma e' \\
(i) & \quad \sigma e' \Downarrow^{\ell} \\
(ii) & \quad e' \approx^I d' \\
(iii) & \quad \lambda x.d' \approx^I t
\end{align*}

by inversion

where BVC stands for Barendregt’s variable convention [3]. Once again, substitution must be performed on $e'$ before the inductive hypothesis is applicable. Substitution
grows the size of $e'$ without altering the number of steps to termination, so the proof is stuck again.

Notice that this obstacle does not exist in a single-stage calculus. In the plain $\lambda$ calculus, $\lambda x. d' \Downarrow$ regardless, and $d' \Downarrow$ does not need to be proved. The root cause of the failure is that $R^0$ blindly closes all free variables when $\lambda^V$ requires direct reasoning on open terms. If I just remove the substitution, however, and say

$$e[R]^J t \iff (\forall \ell. (e, t \in E^\ell \land e \Downarrow^\ell v) \implies (t \Downarrow^\ell u \land v \{R\}^{I, \ell} u)) \quad e, t \text{ can be open}$$

then the distinction between variables will be lost. For example, $x[R]^J y$ holds for arbitrary $x, y$ because $x$ diverges. The right solution is to quantify over substitutions but to allow the substitutions to leave some variables free. The substitutions no longer ensure closedness, so there is no point in keeping the distinction between relations on open terms and those on closed terms. These insights lead to a monolithic formulation of applicative bisimulation.

$$v \{R\}^{M,0} u \overset{\text{def}}{\iff} (v = \tau \overline{e_i} \implies (u = \tau \overline{t_i} \land \overline{e_i} R \overline{t_i}))$$

$$v \{R\}^{M,\ell+1} u \overset{\text{def}}{=} v Ru$$

$$e [R]^M t \overset{\text{def}}{=} \forall \sigma : \text{Var}^{\text{fin}} \rightarrow A. \forall \ell. \quad (\sigma e, \sigma t \in E^\ell \land \sigma e \Downarrow^\ell v) \implies (\sigma t \Downarrow^\ell u \land v \{R\}^{M, \ell} u)$$

$$(\preceq)^M \overset{\text{def}}{=} \max [\preceq]^M$$

$$(\succeq)^M \overset{\text{def}}{=} (\succeq)^M \cap (\preceq)^M$$

Note the direct correspondence with Definition 5.3. A judgment $e \preceq^M t$ requires $t$ to be more terminating than $e$ under arbitrary substitutions, and if both terminate then $\{R\}^\ell$ compares return values after an experiment. However, note the crucial
difference from Definition 5.3 that an experiment does not substitute an argument for the parameter variable of a level-0 abstraction but simply removes the binder. In other words, this monolithic relation does not exploit well-leveled bindings and is thus incomplete. The following definition solves this problem by comparing function bodies by a slightly different relation than the expressions that yielded those functions.

**Notation.** Let $X, Y$ range over the set $\wp_{\text{fin}} \text{Var}$ of all finite subsets of $\text{Var}$. This notation applies to the family notation, so $R_X$ means $R_{X \in \wp_{\text{fin}} \text{Var}}$. Let the signature $\sigma : X | \text{Var} \xrightarrow{\text{fin}} A$ mean that $\sigma : \text{Var} \xrightarrow{\text{fin}} A$ and $\text{dom } \sigma \supseteq X$, i.e. $\sigma$ substitutes for at least the variables in $X$.

**Definition 5.24** (Indexed Applicative Bisimulation). Define the *indexed applicative bisimulation* family of relations $(\sim_X)$ and auxiliary relations as follows. All relations are defined on open terms.

\[
\begin{align*}
  v \{R_X\}^0 u & \iff \\
  v = \lambda x.e & \implies (u = \lambda x.t \land e \mathrel{R\{x\}} t) \\
  v = \langle e \rangle & \implies (u = \langle t \rangle \land e \mathrel{R\emptyset} t)
\end{align*}
\]

\[
v \{R_X\}^{\ell+1} u \iff v \mathrel{R_{\emptyset}} u
\]

\[
e \mathrel{[R\{X\}]} t \iff \forall \sigma : X | \text{Var} \xrightarrow{\text{fin}} A. \forall \ell.
\]

\[
(\sigma e, \sigma t \in E^\ell \land \sigma e \downarrow^\ell v) \implies (\sigma t \downarrow^\ell u \land v \{R_X\}^\ell u)
\]

\[
(\preceq_X) \iff \max (\preceq_X)
\]

\[
(\sim_X) \iff (\preceq_X) \cap (\preceq_X)
\]

Note that \(\{\cdot\}^\ell\) maps a family of relations to a single relation, whereas $\overline{\{\cdot\}}_X$ maps a family to a family.

To exploit well-leveled bindings, I index each relation by the set of variables that are known to be well-leveled. The indexed relation only considers those $\sigma$'s that substitute for all variables in this index set. This strategy is similar to how closing
substitutions are used to eliminate all free variables in the plain \(\lambda\) calculus. In \(\lambda^V\), substituting away all free variables is unsound, so instead only well-leveled variables are substituted away. The set of well-leveled variables is grown precisely when \(\{R\}^\ell\) opens up a binder to compare function bodies, which happens only for level-0 abstractions. A variable bound by a level-0 abstraction cannot be used in an ill-leveled manner because at every point in the body of such an abstraction, the number of brackets around that point is by definition no less than the number of escapes. The relation \((\sim_0)\) at the empty index directly corresponds to the proof principle that I presented in the previous section.

The reader may wonder why I did not adopt another approach instead that avoids indexing the relation into a whole family of relations. Namely, it seems that instead of indexing, the monolithic relation can be fixed by modifying \(\{R\}^{M,0}\) as

\[
v \{R\}^{M',0} u \overset{\text{def}}{\iff} \begin{cases} v = \lambda x. e \implies (u = \lambda x. t \land \forall a. ([a/x] e) R ([a/x] t)) \\ v = \langle e \rangle \implies (u = \langle t \rangle \land e R t) \end{cases}
\]

and updating the other relations to use this definition instead of \(\{R\}^{M,0}\):

\[
v \{R\}^{M',\ell+1} u \overset{\text{def}}{\iff} v R u
\]

\[
e [R]^{M'} t \overset{\text{def}}{\iff} \forall \sigma : Var^\mathrm{fin} \to A. \forall \ell.
\]

\[
(\sigma e, \sigma t \in E^\ell \land \sigma e \Downarrow^\ell v) \implies (\sigma t \Downarrow^\ell u \land v \{R\}^{M',\ell} u)
\]

\[
(\preceq)^{M'} \overset{\text{def}}{=} \max \{\preceq\}^{M'}
\]

\[
(\sim)^{M'} \overset{\text{def}}{=} (\succeq)^{M'} \cap (\preceq)^{M'}
\]

This approach is attractive as it exploits well-leveled bindings without requiring the more complex scheme of mutual coinduction. The problem with this definition is that I cannot seem to find a precongruence candidate that is compatible with this modified monolithic relation while simultaneously being reflexive and closed under substitution.
Recall that the precongruence candidate is required to be closed under substitution in order to handle application. A detailed explanation of the problem was given for the plain \(\lambda\) calculus in the previous section following Lemma 5.5; in the case of \(\lambda^V\), we must prove \([a/x]e' \lessapprox_{M'} [b/x]t'\) given
\[
e = (\lambda x.e') \ a \leadsto_0 [a/x]e' \quad \text{where} \quad a = \lambda y.a' \\
t = (\lambda x.t') \ b \leadsto_0 [b/x]t' \quad \text{where} \quad b = \lambda y.b'
\] (17)
The \(a\) and \(b\) must be abstractions and not code values to exhibit the problem I am about to show. With the modified \([R]^{M'}\) defined in terms of \(\{R\}^{M',0}\), the inductive hypothesis guarantees
\[
\begin{align*}
\text{(i)} \quad & \forall c. \ [c/x]e' \lessapprox_{M'} [c/x]t' \\
\text{(ii)} \quad & \forall c. \ [c/x]a' \lessapprox_{M'} [c/x]b' 
\end{align*}
\] (18)
In order to convert (ii) to \(a' \lessapprox_{M'} b'\), the precongruence candidate must adopt a new rule:
\[
\dfrac{
\forall a. \ [a/x]e^0 \bar{R}^{M'}[a/x]t \quad (\lambda x.d)Rt
}{\overline{(\lambda x.e^0)R^t} \bar{R}^{M'}t}[\text{ALT-PC-Abs}]
\]
or something along this line, and HPC-Ind must be modified to avoid clashing with the new rule:
\[
\dfrac{
x \bar{R}^t \quad \overline{\tau \overline{d_i}} \bar{R}^{M'} \overline{d_i} \quad (\tau \overline{d_i})R^t \quad \tau \overline{e_i} \neq \lambda x.e^0
}{x \bar{R}^{M'}t}[\text{ALT-PC-Var}]
\]
\[
\dfrac{
\overline{(\tau \overline{e_i})R^t} \bar{R}^{M'}t
}{\tau \overline{e_i} \bar{R}^{M'}t}[\text{ALT-PC-Ind}]
\]
The ALT-PC-Ind rule applies only when ALT-PC-Abs does not; without this provision, (17.i) would not be guaranteed. But this updated precongruence candidate fails reflexivity: a derivation of \(\lambda x.x \lessapprox_{M'} \lambda x.x\) using ALT-PC-Abs entails an \(a \text{ priori}\) derivation of \(\lambda x.x \lessapprox_{M'} \lambda x.x\) itself, because the quantification over \(a\) covers the case

\(^1\)Perhaps it is possible to derive ALT-PC-Abs from ALT-PC-Var and ALT-PC-Ind, but I do not see how.
\( a = \lambda x.x \). The ALT-PC-IND rule is thus defunct under an inductive interpretation. If the rules are interpreted coinductively, then reflexivity is recovered but closure under substitution no longer seems to be provable because there is nothing to induct on. In general, proving a property that holds over all members of a coinductive set can be very difficult, and closure under substitution appears to be such a property.

After much trial and error, I have found that in order to prove both reflexivity and closure under substitution, I need to strategically insert auxiliary assumptions of the forms

\[ \forall \sigma \text{ of a certain kind. } (\sigma e)R(\sigma t) \quad \text{and} \]
\[ \forall \sigma \text{ of a certain kind. } (\sigma e)\hat{R}M(\sigma t). \]

The indexed families can be seen as a concise encoding of these assumptions which makes the proofs nearly trivial, as shown in the next section.

### 5.4 Soundness and Completeness of Indexed Applicative Bisimulation

In this section, I will show that the indexed applicative bisimulation relation is sound and complete with respect to \( \approx \). The exact statement of this fact needs some care, as indexed bisimulation is a family of relations while observational equivalence is a single relation. I will first prove that indexed applicative bisimulation coincides with a similarly indexed generalization of \( \approx \):

**Definition 5.25 (Indexed Observational Order and Equivalence).** Define \( e \approx X t \overset{\text{def}}{\iff} \forall \sigma : X|\text{Var} \to \text{A. } \sigma e \approx \sigma t \) and \( e \preceq X t \overset{\text{def}}{\iff} \forall \sigma : X|\text{Var} \to \text{A. } \sigma e \preceq \sigma t \).

After showing \( (\preceq_X) = (\subseteq_X) \), I prove that \( (\subseteq_\emptyset) = (\subseteq) \). The latter step involves the RED-\( E_V \) rule, the main improvement of \( \lambda^V \) over \( \lambda^U \). With these results, Lemma 5.5,
which claims the legitimacy of the extensional reasoning principle, can be proved.

5.4.1 Soundness

The general approach to proving the soundness of indexed applicative bisimulation closely follows Howe’s, as described in Subsection 5.3.1. I define an indexed precongruence candidate that respects contexts and show that it coincides with indexed bisimulation.

**Definition 5.26** (Indexed Precongruence Candidate).

\[
\begin{align*}
\frac{xR_X t}{x \overline{R}_X t} & \quad \text{[IPC-VAR]} \\
\frac{\overline{e_i}R_X \overline{d_i}}{(\tau \overline{d_i})R_X t} (\tau \overline{e_i} \neq \lambda x. e^0) & \quad \text{[IPC-IND]} \\
\frac{e^0 \overline{R}_X d}{(\lambda x. d)R_{X \setminus \{x\}} t} & \quad \text{[IPC-Abs]}
\end{align*}
\]

The rule IPC-IND applies when the left-hand side is not a level-0 abstraction. Level-0 abstractions are handled by IPC-Abs instead. This definition is crafted so that important properties transfer from Howe’s original definition, which I summarize in the following proposition. Most of the assertions are straightforward adaptations from Howe [15]. Monotonicity in \(X\) (Proposition 5.28 (v)) is the only one that has no analogue in [15]; it is needed to convert \(e \lesssim_{X \setminus \{x\}} t\) to \(e \lesssim_X t\) in some parts of the proof.

**Definition 5.27.** An indexed family of relations \(\overline{R}_X\) respects contexts with diminishing indices iff \(\overline{e_i}R_X \overline{e_i} \Rightarrow (\tau \overline{e_i})R_Y (\tau \overline{e_i})\) where \(Y = X \setminus \{x\}\) if \(\tau \overline{e_i} = \lambda x. e^0\) and \(Y = X\) otherwise.
Proposition 5.28 (Basic Facts About Indexed Precongruence). Let $\overline{R_X}$ be a family of preorders that is monotone in $X$, i.e. each $R_X$ is a preorder and $X \subseteq Y \implies R_X \subseteq R_Y$. Then

(i) $\overline{R_X}$ is reflexive for every $X$.

(ii) $\overline{R_X}$ respects contexts with diminishing indices.

(iii) $e\overline{R_X}dR_Xt \implies e\overline{R_X}t$ at each $X$.

(iv) $\overline{R_X} \subseteq \overline{R_X}$.

(v) $\overline{R_X}$ is monotonic in $X$.

Proof.

(i) Trivial induction on $e$ shows $e\overline{R_X}e$.

(ii) By reflexivity of $R_X$, derivation rules for $\overline{R_X}$ subsume this assertion.

(iii) Straightforward induction on $e$ using (i) and transitivity of $R_X$.

(iv) Apply (i) to (iii).

(v) Straightforward induction on $e$ using monotonicity of $\overline{R_X}$ shows

$$(e\overline{R_X}t \land X \subseteq Y) \implies e\overline{R_Y}t.$$ \qed

As noted in the previous section, the crucial property of closure under substitution holds, except that the statement accounts for the indexing.

Definition 5.29. A family of relations $\overline{R_X}$ is closed under substitution with diminishing indices iff $e \overline{R_X} t \land a \overline{R_X\backslash\{x\}} b \implies [a/x]e \overline{R_X\backslash\{x\}} [b/x]t$. 


Lemma 5.30. If $R_X$ is a family of relations, then

\[ (i) \quad e[R]_X t \implies \forall \sigma : X \varfin A. \sigma e[R]_{X \setminus \text{dom} \sigma} t \quad \text{and} \]
\[ (ii) \quad \forall Y \subseteq X. ((\forall \sigma : Y \varfin A. \sigma e[R]_{X \setminus \text{dom} \sigma} t) \implies e[R]_X t). \]

In particular, $e \preceq_X t \iff \forall \sigma : X \varfin A. \sigma e \preceq_\emptyset t$.

Proof. Intuitively, this lemma holds because the set of substitutions constructed by composing $\sigma : Y \varfin A$ and $\sigma' : Z \varfin A$ is equivalent to the set of all $\sigma'' : Y \cup Z \varfin A$. See Section A.6 for details. \hfill \Box

Lemma 5.31. The family $(\preceq_X)$ is closed under substitution with diminishing indices, i.e. $e \preceq_X t \wedge a \preceq_X \{x\} b \implies [a/x]e \preceq_X \{x\} [b/x]t$.

Proof. Induction on $e$ using Lemma 5.30 and Proposition 5.28. I handle the most interesting case $e = x$ here. For the remaining cases, see Section A.6.

[If $e = x$]

\[
\begin{align*}
x \preceq_X t & \quad \text{by inversion} \\
[b/x]x \preceq_X \{x\} [b/x]t & \quad \text{by Lemma 5.30} \quad (1) \\
[a/x]x \preceq_X \{x\} [b/x]x & \quad \text{because } a \preceq_X \{x\} b \text{ by assumption} \quad (2) \\
[a/x]x \preceq_X \{x\} [b/x]t & \quad \text{by Proposition 5.28 (iii) and (1)(2)} \quad \Box
\end{align*}
\]

Remark 5.32. Lemmas 5.30 and 5.31 force Definition 5.26 to quantify over $\sigma : X \varfin A$ instead of $\sigma : X \varfin A_{cl}$. Lemma 5.31 cannot require any of $e, t, a, b$ to be closed in an open-term semantics, and if $[\preceq_X]$ quantifies over $\sigma : X \varfin A_{cl}$ then Lemma 5.30 only works for substitutions that substitute closed terms. Then step (1) would fail in the proof of Lemma 5.31. Perhaps if this restriction can be removed, the strong $\omega$ rule that only considers closed-term arguments can be proved.
I need to prove a few facts about \((\preceq_X)\) that justify applying Proposition 5.28 to 
\((\widehat{\preceq}_X)\). These properties are all intuitively clear, so I will not cite them explicitly in 
later proofs.

**Proposition 5.33.**

(i) \((\preceq_X)\) is reflexive for every \(X\).

(ii) \((\preceq_X)\) is transitive for every \(X\).

(iii) \((\preceq_X)\) is monotonic in \(X\).

**Proof.** The proofs for (i) and (ii) are adapted from [15].

(i) Define \((= X)\) to be syntactic equality for every \(X\). Clearly \((= X) \subseteq [= X]\), so 
by coinduction \((= X) \subseteq (\preceq_X)\) in the product lattice \(\prod_X \varphi(E \times E)\). Therefore,
\[\forall X. (=) \subseteq (\preceq_X).\]

(ii) Define 
\[R \circ S \overset{\text{def}}{=} \{(e, t) : \exists d. eRdS\}.\]
Take any triple \(e, d, t\) such that \(e \preceq_X d \preceq_X t\), and let \(\sigma : X|\Var \overset{\text{fin}}{\rightarrow} A, \ell\) be given. Then \(\sigma e \downarrow^\ell v \implies \sigma d \downarrow^\ell w \implies \sigma t \downarrow^\ell u\) and \(v\{\preceq_X\}^\ell w\{\preceq_X\}^\ell u\). The last assertion is equivalent to \(v\{\preceq_X \circ \preceq_X\}^\ell u\), so 
\[e\{\preceq_X \circ \preceq_X\}t.\]
Then by coinduction \((\preceq_X \circ \preceq_X) \subseteq (\preceq_X)\).

(iii) Suppose \(e \preceq_X t\) and \(X \subseteq Y\). Any \(\sigma : Y|\Var \overset{\text{fin}}{\rightarrow} A\) also satisfies \(\sigma : X|\Var \overset{\text{fin}}{\rightarrow} A\), so if \(\sigma, \sigma t \in E^\ell\) then \(\sigma e \downarrow^\ell v \implies \sigma t \downarrow^\ell u\) where \(v\{\preceq_X\}^\ell u\). Thus \(e \preceq_Y t\).

Now the central lemma that indexed applicative simulation contains the precongruence candidate is ready to be proved. As discussed earlier, this lemma directly implies soundness.

**Lemma 5.34.** \(e \preceq_X t \implies e\{\preceq_X\}t.\)
Proof. Fix a $\sigma$ and an $\ell$, and assume $e \triangleright^\ell v$. Then show $\sigma t \Downarrow^\ell u \land v \{ \bar{e}_X \}^\ell u$ by lexicographic induction on $(n, e)$ with case analysis on the form of $e$. See Section A.6 for details.

**Theorem 5.35** (Soundness of Indexed Applicative Bisimulation). $(\subseteq_X) \subseteq (\approx_X)$, therefore $(\sim_X) \subseteq (\approx_X)$.

**Proof.** Lemma 5.34 and Proposition 5.28 (iv) imply $(\subseteq_X) = (\approx_X)$, so by Proposition 5.28 (ii), it follows that $(\subseteq_X)$ respects contexts with diminishing indices. Suppose $e \subseteq_X t$ and let a $\sigma : X|Var^{\text{fin}}$ and a context $C$ be given such that $C[\sigma e], C[\sigma t] \in E^0$. Then

\[
\sigma e \subseteq_\emptyset \sigma t \quad \text{by Lemma 5.30}
\]
\[
C[\sigma e] \subseteq_\emptyset C[\sigma t] \quad \text{by context-respecting property}
\]
\[
C[\sigma e] \Downarrow^0 \Rightarrow C[\sigma t] \Downarrow^0 \quad \text{by definition of $(\subseteq_X)$}
\]
\[
\sigma e \subseteq_X \sigma t \quad \text{because $C$ is arbitrary}
\]
\[
e \subseteq_X t \quad \text{because $\sigma$ is arbitrary}
\]

Therefore, $(\sim_X) = (\subseteq_X) \cap (\subseteq_X) \subseteq (\approx_X) \cap (\approx_X) = (\approx_X)$.\qed

### 5.4.2 Completeness

To prove completeness, I prove that $(\subseteq_X) \subseteq [\approx_X]$, which by coinduction implies $(\subseteq_X) \subseteq (\approx_X)$. Then if $(\subseteq_\emptyset) = (\approx)$ can be proved, $(\approx) \subseteq (\subseteq_\emptyset)$ will follow. These results transfer to the symmetric reductions, yielding $(\approx_X) \subseteq (\sim_X)$ and $(\approx) \subseteq (\sim_\emptyset)$.

The strategy just outlined hinges on the property that substitution preserves $(\subseteq)$. The proof of $(\subseteq_X) \subseteq [\approx_X]$ requires this property, and the second step, $(\subseteq_\emptyset) = (\approx)$, states more directly that substitution preserves $(\subseteq)$. This property is obvious in the
plain \( \lambda \) calculus because given a substitution \( \sigma = [\overline{a_i/x_i}] \) and an inequivalence \( e \Leftrightarrow t \),

\[
\sigma e = [a_i/x_i] e \Leftrightarrow (\lambda \overline{\overline{a_i}}. e) \overline{\overline{a_i}} \Leftrightarrow (\lambda \overline{\overline{a_i}}. t) \overline{\overline{a_i}} \Leftrightarrow [a_i/x_i] t = \sigma t.
\]

(*19)

Then one only needs to observe that \((\Leftrightarrow) \subseteq (\approx) \subseteq (\lessgtr)\). This proof does not work as it stands in \( \lambda^V \) because \( \beta \) substitution is only sound if \( e, t \in E^0 \) as seen in Subsection 3.3.1.

At level \( \ell > 0 \), this argument is completed by RED-E\(_V\). In fact, this completeness proof was the original motivation for contemplating \( \lambda^V \) instead of \( \lambda^U \). Given \( e \lessgtr t \), the idea is to surround \( e \) and \( t \) with brackets repeatedly until both sides of the inequivalence become level 0. Then I can attach the substitution via \( \beta \) reduction as in the plain \( \lambda \) calculus. Finally, I discharge the extra brackets by attaching escapes and eliminating them by RED-E\(_V\). This maneuver heavily relies on surrounding both sides of an inequivalence \( e \lessgtr t \) by a suitable context, which is justified by the following lemma.

**Lemma 5.36.** The relation \((\lessgtr)\) respects contexts.

*Proof.* Obvious from definition. \( \Box \)

Now, behold the power of RED-E\(_V\):

**Proposition 5.37.** \( \forall \sigma : \text{Var}^\text{fin} \rightarrow A. e \lessgtr t \implies \sigma e \lessgtr \sigma t. \)

*Proof.* Take any \( \ell \) such that \( e, t \in E^\ell \). Let \( \sigma \overset{\text{def}}{=} [a_i/x_i] \).

\[
\langle \langle \cdots (e) \cdots \rangle \rangle \overset{\ell \text{ times}}{\lessgtr} \langle \langle \cdots (t) \cdots \rangle \rangle \quad \text{adding brackets to both sides}
\]

\[
(\lambda \overline{\overline{a_i}}. \langle \langle \cdots (e) \cdots \rangle \rangle) \overline{\overline{a_i}} \overset{\ell \text{ times}}{\lessgtr} (\lambda \overline{\overline{a_i}}. \langle \langle \cdots (t) \cdots \rangle \rangle) \overline{\overline{a_i}} \quad \text{adding applications to both sides}
\]
Noting that both sides of the inequivalence are at level 0,

\[
\langle \ldots \langle \sigma e \rangle \ldots \rangle \ell \text{ times} \lesssim \langle \ldots \langle \sigma t \rangle \ldots \rangle \ell \text{ times}
\]

by RED-\(\beta\)

\[
\langle \ldots \langle \sigma e \rangle \ldots \rangle \ell \text{ times} \lesssim \langle \ldots \langle \sigma t \rangle \ldots \rangle \ell \text{ times}
\]

adding escapes to both sides

\[
\sigma e \lesssim \sigma t \quad \text{by RED-}\text{E}_V
\]

\(\square\)

**Remark 5.38.** Example 5.8, which proved the soundness of RED-\(E_V\) using the extensional proof principle that I am verifying with applicative bisimulation, does not constitute a circular argument. Firstly, the soundness of the proof principle does not require RED-\(E_V\) (only completeness does), and secondly, the RED-\(E_V\) rule is also verified in Subsection 3.2.2 using Takahashi’s method, independently of the extensional proof principle.

The rest of the completeness proof is straightforward.

**Proposition 5.39.** For every \(X\), \((\lesssim) \subseteq (\lesssim_X)\). In particular, \((\lesssim) = (\lesssim_\emptyset)\). Additionally, \((\approx) \subseteq (\approx_X)\) and \((\approx) = (\approx_\emptyset)\).

**Proof.** If \(e \lesssim t\), then \(\sigma e \lesssim \sigma t\) for all \(\sigma : X | \text{Var} \to A\) by Proposition 5.37, so \(e \lesssim_X t\). Therefore \((\lesssim) \subseteq (\lesssim_X)\). When \(X = \emptyset\), the reverse containment \((\lesssim_\emptyset) \subseteq (\lesssim)\) also holds: the \((\lesssim_\emptyset)\) relation implies \((\lesssim)\) under any substitution, including the empty substitution. Hence \((\lesssim) = (\lesssim_\emptyset)\). The statements regarding \((\approx)\) and \((\approx_X)\) follow directly. \(\square\)

**Theorem 5.40** (Completeness of Indexed Applicative Bisimulation). The equality \((\lesssim_X) = (\lesssim_X)\) holds, and so does \((\approx_X) = (\approx_X)\).

**Proof.** By Theorem 5.35, only \((\lesssim_X) \subseteq (\lesssim_X)\) and \((\approx_X) \subseteq (\approx_X)\) need to be proved.
Suppose $e \preceq_X t$ and fix a $\sigma : X|\text{Var} \xrightarrow{\text{fin}} A$ and an $\ell$. By definition $\sigma e \preceq \sigma t$ so $\sigma e \Downarrow^\ell v \implies \sigma t \Downarrow^\ell u$; I will show that if these $v, u$ exist then $v \{\preceq_X\}^\ell u$.

[If $\ell > 0$] Because $v \approx \sigma e \preceq \sigma t \approx u$, by Proposition 5.39 it follows that $v \preceq u$.

[If $\ell = 0$] Split cases by the form of $v$.

[If $v = \lambda x.e'$] If $u$ were of the form $\langle d \rangle$, then the context $\langle \bullet \rangle$ would distinguish $v$ and $u$ because $\langle \lambda x.e' \rangle$ is stuck while $\langle \langle d \rangle \rangle \Downarrow^0 \langle d \rangle$. Therefore, $u = \lambda x.t'$ for some $t' \in E^0$. For any $a \in A$, the equivalence $v \preceq a$ guarantees $[a/x]e' \preceq v a \preceq u a \preceq [a/x]t'$ so using Proposition 5.39, $e' \preceq_{\{x\}} t'$.

[If $v = \langle e' \rangle$] By the same argument as above, $u = \langle t' \rangle$. Then $e' \preceq \text{run } \langle e' \rangle \preceq \text{run } \langle t' \rangle \preceq t'$, so by Proposition 5.39, $e' \preceq t'$.

It follows that $e \{\preceq_X\} t$, so $\{\preceq_X\} \subseteq \{\preceq_X\}$. By coinduction, $\{\preceq_X\} \subseteq \{\preceq_X\}$. Therefore, $(\approx_X) = (\preceq_X) \cap (\preceq_X) = (\preceq_X) \cap (\preceq_X) = (\sim_X)$ for each $X$. \hfill $\square$

5.4.3 Correctness of Extensional Reasoning Principle

I can finally prove Lemma 5.5, i.e. that the extensional reasoning principle is sound and complete. The statement of the lemma is recalled here.

**Proof of Lemma 5.5 (Extensional Proof Principle).**

*Statement.* If a relation $R \subseteq E \times E$ is preserved by substitution, then

(i) $R$ is increasing iff $R \subseteq (\preceq)$.

(ii) $R$ is decreasing iff $R \subseteq (\preceq)$.

(iii) $R$ is justified iff it is increasing and decreasing.

(iv) $R$ is justified iff $R \subseteq (\approx)$.

*Proof.*
(i) Define $eRx \triangleq \forall \sigma : X | Var^{\text{fin}} \rightarrow A. (\sigma e)R(\sigma t)$ and $(\preceq' X) \triangleq R_X \cup (\preceq X)$. When $R$ is preserved by substitution, $R_\emptyset = R$ holds. In this case, by Theorem 5.40, Definition 5.3 states precisely that $R$ is increasing iff $(\preceq'_\emptyset) \subseteq [(\preceq')]_\emptyset$.

Now, suppose $R$ is increasing, i.e. $(\preceq'_\emptyset) \subseteq [(\preceq')]_\emptyset$, and $e \preceq'_X t$ for some $X$. Then

$$e \preceq_X t \lor eR_X t$$

by definition (1)

so for any given $\sigma : X | Var^{\text{fin}} \rightarrow A$,

$$\sigma e \preceq'_\emptyset \sigma t \lor (\sigma e)R(\sigma t)$$

by (1) using Lemma 5.30 (2)

$$\sigma e \preceq'_\emptyset \sigma t$$

from (2) by definition of $(\preceq'_X)$ (3)

$$\sigma e [(\preceq')]_\emptyset \sigma t$$

because $(\preceq'_X) \subseteq [(\preceq')]_X$ (4)

Since $\sigma$ is arbitrary,

$$e [(\preceq')]_X t$$

by Lemma 5.30 (5)

This argument works for all $e, t$, so

$$(\preceq X) \subseteq (\preceq X)$$

by coinduction (6)

In particular, $R \subseteq (\preceq'_\emptyset) \subseteq (\preceq_\emptyset)$, thus $R \subseteq (\preceq_\emptyset) = (\preceq)$.

Conversely, if $R \subseteq (\preceq_\emptyset)$ then $(\preceq'_\emptyset) = (\preceq_\emptyset) = [(\preceq)]_\emptyset \subseteq [(\preceq')]_\emptyset$, so $R$ is increasing.

Note the last containment uses monotonicity of $[.]_X$.

(ii) Immediate from (i).

(iii) Obvious from Definition 5.3.

(iv) Immediate from (i)–(iii).
Chapter 6

Conclusion and Future Work

Throughout this thesis, I have developed equational properties and proof techniques that help a programmer reason about specific staged programs. In particular, I proved the Erasure Theorem which shows that a staged program is equivalent to its erasure if it can be proved equal to an unstaged form. The CBN version has no side condition whereas the CBV version leaves the proof obligation to justify the reordering of evaluation. With the in-depth explanation of specific techniques for exploiting the Erasure Theorems, these results can give programmers a jump-start on formal verification of staged programs. I have also developed a complementary, extensional proof principle for verifying arbitrary equivalences between staged expressions. This principle can be used to establish equivalences where the equational theory falls short and to further develop the equational theory.

This thesis makes significant progress in the understanding of equational properties of multi-level languages. I have created a toolbox here that is not only able to support sophisticated formal reasoning but also formalizes some intuitive correctness criteria upon which multi-stage programmers can rely. These criteria will help programmers conduct multi-stage programming mostly according to intuition without running into serious correctness issues. At the same time, the side conditions for correctness are clearly delineated in this thesis, so that programmers can be educated better about the pitfalls they must avoid. For example, the use of non-careful (ordinary) reduction as a reasoning device in CBV is one such pitfall. Thus, my results
not only help formal verification but also endorse casual multi-stage programming as a viable approach to writing correct code generators.

Applicative bisimulation successfully transfers extensional reasoning to $\lambda^V$. One remaining issue with this theory is that it cannot handle equivalences that are predicated on closedness. As a case in point, extensionality requires reasoning about open-term arguments when intuition suggests that this requirement might not be strictly necessary. I speculate that reasoning under closing substitutions can be recovered by introducing a more refined indexing scheme, namely a full-fledged type system. In a type-safe extension, one can tell apart variables bound at a higher-level from those bound at the current level. This information can be used to quantify over substitutions that eliminate all present- and lower-level variables but not higher-level variables. Investigating such type-directed reasoning techniques would be a useful and interesting direction for future work.

More broadly, studying the overall effect of a safe type system on the equational properties of the calculus is a sensible extension to the current work. An important property that I suspect will hold under a strong type system is conservativity [23] over the plain $\lambda$ calculus, i.e. all observational equivalences in the plain $\lambda$ calculus between plain, unstaged $\lambda$ terms remain valid in the multi-stage calculus. The failure of $\text{EQ-Var}\beta$ showed that staging is not a conservative extension. Although I have demonstrated that staging conserves a large subset of valid equivalences in the plain $\lambda$ calculus, properly restricting the language to make the extension conservative is worth the effort. Full conservativity implies that all legacy programs written in a single-stage language like OCaml can be safely reused without any modification in the corresponding multi-stage extension such as MetaOCaml. This proposition appears to be widely accepted informally, but no formal proof exists.
I believe the key to conservativity again lies in a safe type system. Intuitively, providing ill-leveled variables is the only kind of interference from a staged context to the execution of an unstaged expression that an unstaged context cannot imitate. As I have mentioned several times, a type system which checks that all bindings are well-leveled should rule out such interferences.

Finally, this thesis is foundational in nature and does not consider applications beyond toy examples. I expect that the Erasure Theorem would be a useful basis for almost all verification tasks, but it is conceivable that some additional higher-level machinery will be necessary to handle the complexity of real-world examples. It remains to be seen whether such additional development becomes necessary or what exactly that might entail.
Bibliography


Appendix A

Proof Details

The main text omits details of proofs that do not help illustrate the core ideas. This chapter fills in those omitted details for nontrivial proofs.

Notation. BVC stands for Barendregt’s variable convention [3], which states that a bound variable is distinct from all free variables. IH stands for inductive hypothesis.

A.1 Proof of $e \Downarrow^\ell \iff [\Omega/x]e \Downarrow^\ell$

This section supplies a detailed proof that an open term $e$ terminates iff $[\Omega/x]e$ does, where $\Omega$ is any closed, divergent level-0 term (Proposition A.4). This result completes the proof of Proposition 3.10. The main idea is to classify a small-step into four forms, and to perform case analysis on those forms. It follows that either the execution of $e$ mirrors that of $[\Omega/x]e$ or eventually the $x$ and $\Omega$ occupy the holes of the evaluation contexts, forcing divergence in both $e$ and $[\Omega/x]e$. This technique of classifying small-steps is fairly standard and appears in a number of places [4, 23, 13, 14].

Lemma A.1 (Classification). Suppose $\sigma : \text{Var} \xrightarrow{\text{fin}} E_0^\text{cl}$ is a substitution that maps variables to closed level-0 expressions. Note that $\text{dom} \sigma = \emptyset$ is allowed. If $\sigma e$ small-steps at level $\ell$, then one of the following conditions hold. In all cases, evaluation contexts are assumed not to bind $x$.*

*Unlike in single-stage languages, an evaluation context can bind variables at level $> 0$. 
Proof. Induction on $e$.

[If $e = x$] Condition (ii) holds with $E_{\ell,m} = \bullet$.

[If $e = e_1 e_2$] Inversion generates three cases, two of which are trivial.

[If $\sigma e_1$ small-steps] Immediate from IH.

[If $\sigma e_1 \in V^\ell$ and $\sigma e_2$ small-steps] Immediate from IH.

[If $e \leadsto d$ is derived by SS-$\beta$] By inversion

\begin{align*}
(i) \; \ell &= 0 \quad (ii) \; \sigma e_1 = \lambda x. e' \quad (iii) \; e' \in E^0 \quad (iv) \; \sigma e_2 \in V^0
\end{align*}

Case analysis on the form of $e_1$ generates two cases: $e_1 = x$ and $e_1 = \lambda x. e'_1$.

[If $e_1 = x$] Condition (iii) holds.

[If $e_1 = \lambda x. e'_1$] Condition (i) holds. Let any $\sigma' : Var^{\text{fin}} \rightarrow E^0_{\text{cl}}$ be given.

\[
\sigma'((\lambda x. e'_1) e_2) = (\lambda x. \sigma' e'_1) \sigma e_2 \quad \text{because } x \not\in \text{dom } \sigma' \text{ by BVC}
\]

\[
[\sigma' e_2/x] \sigma' e'_1 = \sigma'([e'_2/x] e'_1) \quad \text{because } \sigma' \text{ only substitutes closed terms}
\]

\[
\sigma' e \leadsto \sigma'([e'_2/x] e_1) \quad \text{immediately}
\]

[If $e = \lambda x. e'$] $x \not\in \text{dom } \sigma$ by BVC, so $\sigma e = \lambda x. \sigma e'$. By inversion $\sigma e'$ small-steps at level $\ell$, so IH is applicable; the conclusion is then immediate from IH.

[If $e = \langle e' \rangle$] Immediate from IH.

[If $e = \langle e' \rangle$] Inversion generates two cases.

[If $\sigma e'$ small-steps] Immediate from IH.
[If $\sigma e \leadsto_\ell d$ is derived by SS-$E_V$] By inversion,

(i) $\ell = 1$  (ii) $\sigma e' = \langle d \rangle$  (iii) $d \in E^0$.

Case analysis on the form of $e'$ yields two cases.

[If $e' = x$] Condition (iv) holds.

[If $e' = \langle e'' \rangle$ where $\sigma e'' = d$] Condition (i) holds.

[If $e = \text{run } e'$] Similar to the preceding case.

\[ \square \]

**Lemma A.2.** Let $\sigma : \text{Var} \xrightarrow{\text{fin}} \{ e \in E^0 : e \upharpoonright^0 \}$ be a substitution that substitutes divergent level-0 expressions. Then $v \in V^\ell \iff \sigma v \in V^\ell$.

**Proof.** For $\ell > 0$, this lemma is a special case of Lemma 3.11. If $\ell = 0$ then $e$ must be $\lambda x.t^0$ or $\langle t^0 \rangle$, so Lemma 3.11 ensures $\sigma e \in V^0$.

\[ \square \]

**Lemma A.3.** Let $\sigma, \sigma' : \text{Var} \xrightarrow{\text{fin}} \{ e \in E : e \upharpoonright^0 \}$ be substitutions that only substitute divergent level-0 terms. Then $\sigma e \Downarrow^\ell \iff \sigma' e \Downarrow^\ell$ for any $e$.

**Proof.** As the statement is symmetric, I only need to prove one direction. Suppose $\sigma e \leadsto_\ell^n v$ for some $v \in V^\ell$. I will prove $\sigma' e \Downarrow^\ell$ by induction on $n$. If $n = 0$, then by Lemma A.2, $\sigma e \in V^\ell \iff e \iff e \in V^0 \iff \sigma' e \in V^\ell$. If $n > 0$, then I perform case analysis on the first small-step of $\sigma e$ using Lemma A.1. Conditions (iii) and (iv) are vacuous because $\sigma$ does not substitute expressions of the forms $\lambda x.t$ and $\langle t \rangle$. For the remaining cases:

(i) The $\sigma e$ small-steps as $\sigma e \leadsto_\ell \sigma d \leadsto_\ell^{n-1} v$, and $\sigma' e \leadsto_\ell \sigma' d$. By IH $\sigma' d \Downarrow^\ell$, so $\sigma' e \Downarrow^\ell$.

(ii) The $e$ must decompose as $E^{\ell,m}[x]$ where $E^{\ell,m}$ does not bind $x$, and $\sigma x$ small-steps at level $m$. By assumption $\sigma x \in E^0$ so in order for $\sigma x$ to small-step, $m = 0$ is necessary. But then by Lemma 3.6 $\sigma x \Downarrow^0$, contrary to assumption. This case is therefore vacuous.

\[ \square \]
Proposition A.4. If $\Omega$ is a closed, level-0 divergent term, then $e \Downarrow^\ell \iff [\Omega/x]e \Downarrow^\ell$.

Proof. Take $\sigma = \emptyset$ and $\sigma' = [\Omega/x]$ in Lemma A.3. \qed

A.2 Proof Details for Confluence

This section fills in details for the proof of confluence of the reduction relation, which was reduced in the main text to confluence of parallel reduction.

Lemma A.5. If $e, t \in E^0$ and $a, b \in A$ then $(e \nrightarrow t \land a \nrightarrow b) \implies [a/x]e \nrightarrow [b/x]t$.

Proof. Induction on the parallel reduction $e \nrightarrow t$ with case analysis on the last rule used to derive it. Let $N \overset{\text{def}}{=} n + #(x, t) \cdot m$.

[PR-VAR] $e = y = t$ and $n = 0$ by inversion.

[If $x = y$] $N = m$, so $[a/x]e = a \nrightarrow b = [b/x]t$.

[If $x \neq y$] $N = 0$, so $[a/x]e = y \nrightarrow y = [b/x]t$.

[PR-Abs]

(i) $e = \lambda y.e'$  (ii) $t = \lambda y.t'$  (iii) $e' \nrightarrow t'$  

by inversion \hspace{2cm} (1)

$\#(x, t) = \#(x, t')$. \hspace{2cm} because $x \neq y$ by BVC \hspace{2cm} (2)

$N = n + #(x, t') \cdot m$ \hspace{2cm} by (2), defn of $N$ \hspace{2cm} (3)

$[a/x]e' \nrightarrow [b/x]t'$ \hspace{2cm} by IH on (1.iii) with (3) \hspace{2cm} (4)

$\lambda y.[a/x]e' \nrightarrow^N \lambda y.[a/x]t'$ \hspace{2cm} by PR-Abs

Therefore $[a/x]\lambda y.e' \nrightarrow^N [a/x]\lambda y.t'$ because $x \neq y$.

[PR-App]

(i) $e = e_1 \cdot e_2$  (ii) $t = t_1 \cdot t_2$  

(iii) $e_i \nrightarrow t_i$ ($i = 1, 2$)  (iv) $n = n_1 + n_2$  

by inversion
\[ [a/x]e_i^{n_i + \#(x, t_1) \cdot m} \gg [b/x]t_i \quad (i = 1, 2) \] by IH on (iii)

\[ [a/x](e_1 \ e_2) \overset{N}{\gg} [b/x](t_1 \ t_2) \] by PR-App

where the last step uses the fact that

\[ n_1 + \#(x, t_1) \cdot m + n_2 + \#(x, t_2) \cdot m = n_1 + n_2 + \#(x, t_1 \ t_2) \cdot m = N. \]

\[ \text{[PR-}\beta\text{]} \]

\[
\begin{align*}
(i) \quad & e = (\lambda y. d) \ c \in E^0 & \quad (ii) \quad & t = [c'/y]d' \\
(iii) \quad & d \gg d' & \quad (iv) \quad & c \overset{n_c}{\gg} c' \\
(v) \quad & n = n_d + \#(y, d') \cdot n_c + 1
\end{align*}
\]

\[
\begin{align*}
[a/x]d^{n_d + \#(x, d') \cdot m} \gg [b/x]d' & \quad \text{by IH on (iii)} \quad (5) \\
[a/x]c^{n_c + \#(x, c') \cdot m} \gg [b/x]c' & \quad \text{by IH on (iv)} \quad (6)
\end{align*}
\]

Taking

\[ M \overset{\text{def}}{=} (n_d + \#(x, d') \cdot m) + \#(y, d') \cdot (n_c + \#(x, c') \cdot m) + 1 \]

and noting that \( x \neq y \) by BVC,

\[
\begin{align*}
[a/x]((\lambda y. d) \ c) \overset{M}{\gg} [[b/x]c'/y][b/x]d' & \quad \text{by PR-}\beta_\nu \text{ on (5)(6)} \\
[[b/x]c'/y][b/x]d' = [b/x][c'/y]d' & \quad \text{because } y \not\in FV(b) \text{ by BVC} \\
[a/x]((\lambda y. d) \ c) \overset{M}{\gg} [b/x][c'/y]d' & \quad \text{by the two preceding lines}
\end{align*}
\]

as required. For the complexity,

\[
M = (n_d + \#(x, d') \cdot m) + \#(y, d') \cdot (n_c + \#(x, c') \cdot m) + 1
\]

\[
= n_d + \#(y, d') \cdot n_c + 1 + (\#(x, d') + \#(y, d') \#(x, c')) \cdot m
\]

\[
= n + \#(x, [c'/y]d') \cdot m
\]
as required.

[Other cases] Immediate from IH. □

**Lemma A.6.** If $v \in V^0$ and $v \rightarrow^* e$, then $v$ and $e$ have the same form: $v = \lambda x.t \iff e = \lambda x.d$ and $v = \langle t \rangle \iff e = \langle d \rangle$.

**Proof.** $e \in V^0$ follows from Lemma 3.13. If $v$ is $\lambda x.t$ or $\langle t \rangle$ for some $t \in E^0$, then $e$ is $\lambda x.d$ or $\langle d \rangle$, respectively, where $t \rightarrow^* d$ because reduction does not remove $\lambda$ or $\langle \cdot \rangle$ in head position. This implication can be reversed because given $v \in V^0$, these two cases are exhaustive and mutually exclusive. By level preservation $d \in E^0$ so $e$ is a level 0 value of the same form as $v$. □

**Lemma A.7.** If $a \rightarrow^* e$ then $e \in A$.

**Proof.** Immediate from Lemma 3.12 for CBN and from Lemma 3.13 for CBV. □

**Proof of Lemma 3.18** (Takahashi’s Property).

**Statement.** $e \gg t \implies t \gg e^*$.

**Proof.** Induction on $e$ with case analysis on $e$.

[If $e = x$] $x \gg x = t = x^*$.

[If $e = \lambda x.e'$]

\[
\begin{align*}
(i) \quad & t = \lambda x.t' \\
(ii) \quad & e' \gg t' & \text{by inversion}
\end{align*}
\]

\[
\begin{align*}
& t' \gg (e')^* & \text{by IH on (ii)}
\end{align*}
\]

\[
\begin{align*}
& \lambda x.t' \gg \lambda x.(e')^* & \text{by PR-Abs}
\end{align*}
\]

and $\lambda x.(e')^*$ is just $e^*$.

[If $e = (\lambda x.d) a$ where $d \in E^0$]

\[
\begin{align*}
(i) \quad & t = (\lambda x.d') a' \vee t = [a'/x]d' \\
(ii) \quad & d \gg d' & \text{by inversion}
\end{align*}
\]

\[
\begin{align*}
(iii) \quad & a \gg a' & \text{(1)}
\end{align*}
\]
where the shape of $t$ depends on whether the last rule used to derive $e \gg t$ is PR-App or PR-β.

(i) $d' \gg d^*$  (ii) $a' \gg a^*$  
by IH on (1.ii)(1.iii)  

(ii) $a^* \in A$  
using Lemma A.7 on (2)  

$[a^*/x]d^* = ((\lambda x.d) a)^*$  
by PR-β on (2)(3)  
and definition of $\cdot^*$

where the last step needs Lemma A.5 as well if $t = [a'/x]d'$.

[If $e = e_1 e_2$ and $e$ is not a β redex]

(i) $t = t_1 t_2$  (ii) $e_i \gg t_i$ ($i = 1, 2$)  
by inversion

$t_i \gg e_i^*$ ($i = 1, 2$)  
by IH on (ii)

$t \gg e_1^* e_2^* = e^*$.  
by PR-App and definition of $\cdot^*$

[If $e = \langle e' \rangle$]

(i) $t = t' \vee t = \langle t' \rangle$  (ii) $e' \gg t'$  
by inversion

$t' \gg (e')^*$  
by IH on (ii)

$\langle t' \rangle \gg (e')^* = (\langle e' \rangle)^*$  
by PR-E and definition of $\cdot^*$

[If $e = \langle e' \rangle$ where $e' \in E^0$]  Similar to the preceding case.

[If $e = \langle e' \rangle$ or $e = \langle e' \rangle^*$ or $e = \text{run } e'$, and $e$ is not a redex]  Immediate from IH.

**Proof of Proposition 3.19.**

Statement. $(\gg^*)$ is confluent: more specifically, if $e_1 \ll^n e \gg^k e_2 \Rightarrow \exists e'. e_1 \gg^k e' \ll^n e_2$.

Proof. Induction on $(n, k)$ under lexicographical ordering.
[If $n = 0$] $e = e_1$, so take $e' \overset{\text{def}}{=} e_2$.

[If $k = 0$] $e = e_2$, so take $e' \overset{\text{def}}{=} e_1$.

[If $n, k > 0$]

$$
\exists e_1', e_1 \ll e_1' \ll^{n-1} e \gg^k e_2 \quad \text{because } n > 0
$$

$$
\exists e_3', e_1' \gg^k e_3 \ll^{n-1} e_2 \quad \text{by IH} \quad (1)
$$

$$
\exists e_3', e_1 \ll e_1' \gg^{k-1} e_3' \gg e_3 \quad \text{by (1) and } k > 0 \quad (2)
$$

$$
\exists e_4', e_1 \gg^{k-1} e_4 \ll e_3' \quad \text{by IH} \quad (3)
$$

$$
e_4 \ll e_3' \gg e_3 \quad \text{by (2)(3)}
$$

$$
e_4 \gg (e_3')^* \ll e_3 \quad \text{by Takahashi’s property} \quad (4)
$$

$$
e_1 \gg (e_3')^* \ll^n e_2 \quad \text{by (1)(3)(4)} \quad \square
$$

### A.3 Proof Details of Standardization

This section provides proof details of Theorem 3.20, which is traditionally proved via a “standardization” lemma and is therefore sometimes called standardization itself by abuse of terminology. Takahashi’s method obviates the need to define an auxiliary standard reduction, however.

**Proof of Lemma 3.23** (Transition).

**Statement.** If $e \in E^\ell$ and $v \in V^\ell$ then $e \gg^n v \implies \exists u \in V^\ell. e \mapsto^* u \gg v$.

**Proof.** If $e \in V^\ell$ the conclusion is obvious, so assume $e \notin V^\ell$. Lexicographically induct on $(n, e)$, with case analysis on the last rule used to derive the parallel reduction. Note that before invoking IH on a sub-judgment $e' \gg t$ of $e \gg v$, where $n' \leq n$ and $e'$ is a subterm of $e$, the side conditions $e' \in E^k$ and $t \in V^k$ must be checked (where $k$ is $\ell$ or $\ell \pm 1$ depending upon the shape of $e$). If $k > 0$, checking the levels of $e'$ and $t$
suffice; otherwise, their shapes must be analyzed.

[PR-VAR] Vacuous: $x \not\in V^\ell$ so $v \neq x$.

[PR-Abs]

(i) $e = \lambda x. t$  (ii) $v = \lambda x. t'$  (iii) $t \xrightarrow{n} t'$ by inversion  

(i) $\ell > 0$  (ii) $t \in E^\ell$ because $\lambda x. t \in E^\ell \setminus V^\ell$  

$t' \in V^\ell$ because $\lambda x. t' \in V^\ell$ and $\ell > 0$  

(2.ii) and (3) justify using IH on (1.iii).

(i) $t \overset{\ell}{\sim}^* u \gg t'$  (ii) $u \in V^\ell$ by IH on (1.iii)  

$\lambda x. t \overset{\ell}{\sim}^* \lambda x. u \gg \lambda x. t'$ from (4.i)  

where $\lambda x. u \in V^\ell$ because $u \in V^\ell$ and $\ell > 0$.

[PR-App]

\[
\begin{align*}
\{ \text{(i) } e & = e_1 e_2 \quad \text{(ii) } v = v_1 v_2 \} \quad \text{by inversion} \quad (5) \\
\text{(iii) } e_i \xrightarrow{n_i} v_i \text{ (} i = 1, 2 \text{) } \quad \text{(iv) } n = n_1 + n_2 \quad \\
\text{(i) } \ell > 0 \quad \text{(ii) } v_1, v_2 \in V^\ell \quad \text{because } v \in V^\ell \quad (6) \\
e_1, e_2 \in E^\ell \quad \text{because } e \in E^\ell \quad (7)
\end{align*}
\]

(6.ii) and (7) justify using IH on (5.iii).

(i) $e_i \overset{\ell}{\sim}^* w_1 \gg v_i \text{ (} i = 1, 2 \text{) } \quad \text{(ii) } w_1, w_2 \in V^\ell \quad \text{by IH on (5.iii)} \quad (8) \\
e_1 e_2 \overset{\ell}{\sim}^* w_1 e_2 \overset{\ell}{\sim}^* w_2 \gg v_1 v_2 \quad \text{by (8.i)}
\]

where $w_1, w_2 \in V^\ell$ by (6.i) and (8.ii).
\[ \begin{align*}
\text{PR-}\beta & \\
(\text{i}) \ e &= (\lambda x.d) \ a & (\text{ii}) \ v &= [a'/x]d' \\
(\text{iii}) \ d \overset{n_1}{\gg} d' & \quad (\text{iv}) \ a \overset{n_2}{\gg} a' & \text{by inversion} \\
(\text{v}) \ n &= n_1 + \#(x, d') \cdot n_2 + 1
\end{align*} \]

\[ \begin{align*}
(\text{i}) \ \ell &= 0 & (\text{ii}) \ d &\in E^0 & \text{because } e \in E^0 \setminus V^\ell \\
a' &\in A & \text{by Lemma A.7 and (9.iv)} & \text{(11)} \\
[a/x]d \overset{n-1}{\gg} [a'/x]d' & \text{by Lemma A.5 and (11)} & \text{(12)} \\
[a/x]d &\in E^0 & \text{by Lemma 3.11 and (10.ii)} & \text{(13)}
\end{align*} \]

IH can be invoked on (12) because (13) and \([a'/x]d' = v \in V^0\).

\[ \begin{align*}
(\text{i}) \ [a/x]d &\leadsto^0_\ast u \gg [a'/x]d' & (\text{ii}) \ u &\in V^0 & \text{by IH} & \text{(14)} \\
e' &\leadsto [a/x]d \leadsto^0_\ast u \gg [a'/x]d' & \text{by SS-}\beta
\end{align*} \]

\[ \text{[PR-BRK, PR-ESC, or PR-RUN]} \quad \text{All of these cases are similar. PR-BRK is worked out here as an example.} \]

\[ \begin{align*}
(\text{i}) \ e &= \langle e' \rangle & (\text{ii}) \ v &= \langle v' \rangle & (\text{iii}) \ e' &\overset{n}{\gg} v' & \text{by inversion} & \text{(15)} \\
e' &\in E^{\ell+1} & \text{because } \langle e' \rangle \in E^\ell & \text{(16)} \\
v' &\in V^{\ell+1} & \text{because } \langle v' \rangle \in V^\ell & \text{(17)}
\end{align*} \]

(16)(17) justify using IH on (15.iii).

\[ \begin{align*}
(\text{i}) \ e' &\leadsto^\ast_\ell u' \gg v' & (\text{ii}) \ u' &\in V^{\ell+1} & \text{by IH on (15.iii)} & \text{(18)} \\
\langle e' \rangle &\leadsto^\ast_\ell \langle u' \rangle \gg \langle v' \rangle & \text{from (18.i)}
\end{align*} \]

where \(\langle u' \rangle \in V^\ell\) by (18.ii).
[PR-E]

(i) \( e = \langle e' \rangle \)  \hspace{1cm} (ii) \( e'^{n-1} v \) \hspace{1cm} by inversion \hspace{1cm} (19)

(i) \( \ell > 0 \)  \hspace{1cm} (ii) \( e' \in E^\ell \) \hspace{1cm} because \( \langle e' \rangle \in E^\ell \) \hspace{1cm} (20)

Invoking IH on (19.ii) is justified by (20.ii) and the assumption that \( v \in V^\ell \).

(i) \( e' \sim^* u \gg v \)  \hspace{1cm} (ii) \( u \in V^\ell \) \hspace{1cm} by IH on (19.ii) \hspace{1cm} (21)

\( \langle e' \rangle \sim^* \langle u \rangle \) \hspace{1cm} from (21.i)

Then split cases on \( \ell \).

[If \( \ell = 1 \)]

\( \langle u \rangle \sim^*_1 u \) \hspace{1cm} by \( E_V \) using (21.ii)

\( \langle e' \rangle \sim^*_1 \langle u \rangle \sim^*_1 u \gg v \) \hspace{1cm} immediately

[If \( \ell > 1 \)]

\( \langle e' \rangle \sim^*_1 \langle u \rangle \gg v \) \hspace{1cm} by (21.i)

where

\( \langle u \rangle \in E^{\ell-2} = V^{\ell-1} \) \hspace{1cm} by (21.ii) and \( \ell > 1 \)

\( \langle u \rangle \in V^\ell \) \hspace{1cm} immediately

[PR-R]

(i) \( e = \text{run} \langle e' \rangle \)  \hspace{1cm} (ii) \( e'^{n-1} v \)  \hspace{1cm} (iii) \( e' \in E^0 \) \hspace{1cm} by inversion \hspace{1cm} (22)

\( \ell = 0 \) \hspace{1cm} by \text{run} \langle e' \rangle \notin V^\ell \hspace{1cm} (23)

while (22.iii)
IH on (22.ii) is justified by (22.iii) and the assumption $v \in V^\ell$.

\[
\begin{align*}
(i) \ e' & \leadsto_t^* u \gg v \quad \text{by IH on (22.ii)} \quad (24) \\
\text{run } \langle e' \rangle & \leadsto e' \quad \text{by } R_V \text{ using (22.iii)} \\
\text{run } \langle e' \rangle & \leadsto e' \leadsto_t^* u \gg v \quad \text{immediately}
\end{align*}
\]

**Proof of Lemma 3.24** (Permutation).

**Statement.** If $e, t, d \in E^\ell$ then $e \gg \ell t \leadsto_d \Rightarrow \exists t' \in E^\ell. e \leadsto^+ \ell t' \gg d$.

**Proof.** Induction on $n$ with case analysis on the last rule used to derive the parallel reduction. In all cases but PR-$\beta$, the complexity $n$ obviously diminishes in the IH, so I omit this check. In fact, I omit the complexity annotation altogether as it has no other use in this proof.

[PR-VAR] Vacuous: $e = x = t$ so $t \not\leadsto$. 

[PR-Abs]

\[
\begin{align*}
(i) \ e &= \lambda x.e' \quad (ii) \ t = \lambda x.t' \quad (iii) \ e' \gg t' & \text{by inversion} \\
(i) \ \ell > 0 \quad (ii) \ d = \lambda x.d' \quad (iii) \ t' \leadsto_d & \text{by inversion on } \lambda x.t' \leadsto_d \quad (2) \\
& e' \gg t' \leadsto_d & \text{by (1.iii)(2.iii)} \\
(i) \ e' & \leadsto^+ \ell t'' \gg d' \quad (ii) \ t'' \in E^\ell & \text{by IH} \\
\lambda x.e' & \leadsto^+ \ell \lambda x.t'' \gg \lambda x.d' & \text{from (3.i)}
\end{align*}
\]

[PR-App]

\[
\begin{align*}
(i) \ e &= e_1 e_2 \quad (ii) \ t = t_1 t_2 \quad (iii) \ e_i \gg t_i \ (i = 1, 2) & \text{by inversion} \\
& \text{Inversion on } t_1 t_2 \leadsto_d \text{ generates two cases.}
\end{align*}
\]
[If \( t_1 \) \( t_2 \) \( \overset{\ell}{\rightarrow} d \) is derived by SS-\( \beta \)]

\[
\begin{align*}
(i) \quad \ell &= 0 \\
(ii) \quad t_1 &= \lambda x. t_3 \\
(iii) \quad t_2 &\in A \\
(iv) \quad d &= [t_2/x]t_3
\end{align*}
\]

by inversion \( \quad \) (5)

Noting that \( t_1 \in V^0 \),

\[
\begin{align*}
(i) \quad e_1 &\overset{0}{\sim}^* v_1 \gg t_1 \\
(ii) \quad v_1 &\in V^0 \\
v_1 &= \lambda x.e_3 \\
e_3 &\gg t_3
\end{align*}
\]

by Transition on (5.iii) \( \quad \) (6)

by Lemma A.6 with (5.ii)(6.i) \( \quad \) (7)

by inversion on (6.i) using (5.ii)(7) \( \quad \) (8)

Now split cases by evaluation strategy.

[In CBV] Observing that \( t_2 \in A = V^0 \),

\[
\begin{align*}
(i) \quad e_1 e_2 &\overset{0}{\sim}^* v_2 \gg t_2 \\
(ii) \quad v_2 &\in V^0 \\
e_1 e_2 &\overset{0}{\sim}^* (\lambda x.e_3) v_2 \overset{0}{\rightarrow} [v_2/x]e_3 \\
[v_2/x]e_3 &\gg [t_2/x]t_3 \\
e_1 e_2 &\overset{0}{\sim}^+ [v_2/x]e_3 \gg d
\end{align*}
\]

by (9)(10) \( \quad \) (9)

by (9.1)(7) and SS-\( \beta \) \( \quad \) (10)

by Lemma A.5 using (8)(9.i) \( \quad \) (11)

by (10)(11)(5.iv) \( \quad \) (12)

[In CBN] Observing that \( t_2 \in A = V^0 \),

\[
\begin{align*}
e_1 e_2 &\overset{0}{\sim}^* (\lambda x.e_3) t_2 \overset{0}{\rightarrow} [t_2/x]e_3 \\
[t_2/x]e_3 &\gg [t_2/x]t_3 \\
e_1 e_2 &\overset{0}{\sim}^+ [t_2/x]e_3 \gg d
\end{align*}
\]

by (9.1)(7) and SS-\( \beta \) \( \quad \) (12)

by Lemma A.5 using (8) \( \quad \) (13)

by (12)(13)(5.iv) \( \quad \) (14)

[If \( t_1 \in V^\ell \) but \( t_1 \) \( t_2 \) is not a \( \beta \) redex]

\[
(t_1 \bullet) \in ECtx^{\ell,\ell} \quad \text{because} \ t_1 \in V^\ell \quad \quad \) (14)

\[
(i) \quad d = t_1 d_2 \\
(ii) \quad t_2 \overset{\ell}{\rightarrow} d_2
\]

by Proposition 3.5 and (14) \( \quad \) (15)
\[ e_2 \gg t_2 \leadsto d_2 \quad \text{by (4.iii)(15)} \]

(i) \[ e_2 \leadsto^+ d'_2 \gg d_2 \quad \text{by IH} \quad (16) \]

\[ e_1 \quad e_2 \leadsto^+ e_1 \quad d'_2 \gg t_1 d_2 \quad \text{from (16.i)(4.iii)} \]

[If \( t_1 \notin V^\ell \)]

\[ (\bullet \quad t_2) \in ECtx^\ell \ell \quad \text{clearly} \quad (17) \]

(i) \[ d = d_1 \quad t_2 \quad (ii) \quad t_1 \leadsto d_1 \quad \text{by Proposition 3.5 and (17)} \]

(iii) \[ e_1 \leadsto^+ d'_1 \gg d_1 \quad (iv) \quad d'_1 \in E^\ell \quad \text{by IH} \quad (18) \]

\[ e_1 \quad e_2 \leadsto^+ d'_1 \quad e_2 \gg d_1 \quad t_2 \quad \text{from (18.i)(4.iii)} \]

[PR-\( \beta \)] This is the only case in which the check on complexity is non-trivial.

\[
\begin{align*}
(i) & \quad e = (\lambda x. e') \quad a \\
(ii) & \quad t = [a'/x]e'' \\
(iii) & \quad e' \gg e'' \quad (iv) \quad a \gg a' \\
(v) & \quad e' \in E^0 \\
(vi) & \quad n = n_1 + \#(x,e'') \cdot n_2 + 1
\end{align*}
\]

by inversion \( (19) \)

\[ a' \in A \quad \text{by Lemma A.7 and (19.iv)} \quad (20) \]

\[ [a/x]e'^{n_1+\#(x,e'') \cdot n_2} \gg [a'/x]e'' \quad \text{by Lemma A.5 and } (19.iii)(19.iv)(20) \]

Observe that the complexity is indeed smaller. Noting (19.ii),

\[
\begin{align*}
(i) & \quad [a/x]e' \leadsto^+ t' \gg d \\
(ii) & \quad t' \in E^\ell \quad \text{by IH on (21) and } t \leadsto d \quad (22)
\end{align*}
\]

Now, to connect (22.i) to \( e \):

\[ e \leadsto [a/x]e' \quad \text{by SS-} \beta_v \quad (23) \]
\[ [a/x]e' \in E^0 \quad \text{by (23)} \]
\[ \ell = 0 \quad \text{because (22.i)(24)} \]
\[ e \rightsquigarrow [a/x]e' \rightsquigarrow^0 t' \gg d \quad \text{by (22.i)(23)(25)} \]

**[PR-Esc]**

(i) \( e = \neg e' \)  
(ii) \( t = \neg t' \)  
(iii) \( e' \gg t' \)  
by inversion \hfill (26)

(i) \( \ell > 0 \)  
(ii) \( t', e' \in E^{\ell-1} \)  
because \( \neg e', \neg t' \in E^\ell \) \hfill (27)

Inversion on \( \neg t' \rightsquigarrow d \) generates two cases.

**[If \( \neg t' \rightsquigarrow d \) is derived by SS-E\(_V\)]**

(i) \( \ell = 1 \)  
(ii) \( t' = \langle d \rangle \)  
(iii) \( d \in E^0 \)  
by inversion \hfill (28)

\( t' \in V^0 \)  
by (28.ii)(28.iii) \hfill (29)

(i) \( e' \rightsquigarrow^* v \gg t' \)  
(ii) \( v \in V^0 \)  
by Transition, \hfill (30)

using (26.iii)(29)

(i) \( v = \langle d' \rangle \)  
(ii) \( d' \in E^0 \)  
by Lemma A.6, \hfill (31)

using (28.ii)(28.iii)(29)(30)

\( \neg\langle d'\rangle \rightsquigarrow d' \)  
by SS-E\(_V\) \hfill (32)

\( d' \gg d \)  
by inversion on (30.i) \hfill (33)

\( \neg\langle e'\rangle \rightsquigarrow^* \neg\langle d'\rangle \rightsquigarrow d' \gg d \)  
by (30.i)(32)(31)

**[If \( t' \) small-steps]**

(i) \( d = \neg d' \)  
(ii) \( t' \rightsquigarrow d' \)  
by inversion \hfill (34)

\( e' \gg t' \rightsquigarrow d' \)  
by (26.iii)(33.ii) \hfill (35)

\( e' \rightsquigarrow^+ t' \gg d' \)  
by IH
\[
- e' \xrightarrow{\ell^+} - t' \gg -d'
\]

**[PR-E]**

(i) \( e = \langle e' \rangle \)  (ii) \( e' \gg t \)  

by inversion  \( (34) \)

(i) \( \ell > 0 \)  (ii) \( e' \in E_{\ell}^t \)  

because \( \langle e' \rangle \in E_{\ell}^t \)  \( (35) \)

\( e' \gg t \xrightarrow{\ell} d \)  

by \( (34.ii) \) and assumption  \( (36) \)

(i) \( e' \xrightarrow{\ell} t' \gg d \)  (ii) \( t' \in E_{\ell}^t \)  

by IH, justified by \( (35.ii) \)  \( (36) \)

\( \langle e' \rangle \xrightarrow{\ell^+} -\langle t' \rangle \gg d \)  

using SS-Ctx and PR-E

where \( (36.ii) \) guarantees \( -\langle t' \rangle \in E_{\ell}^t \).

**[PR-Run]**

(i) \( e = \text{run } e' \)  (ii) \( t = \text{run } t' \)  (iii) \( e' \gg t' \)  (iv) \( e', t' \in E_{\ell}^t \)  

by inversion  \( (37) \)

Inversion on \( \text{run } t' \xrightarrow{\ell} d \) generates two cases.

[If \( \text{run } t' \xrightarrow{\ell} d \) is derived by SS-Rv]

(i) \( \ell = 0 \)  (ii) \( t' = \langle d \rangle \)  (iii) \( d \in E_0^t \)  

by inversion  \( (38) \)

(i) \( e' \xrightarrow{0^*} v \gg \langle d \rangle \)  (ii) \( v \in V_0^t \)  

by Transition, using  \( (39) \)

\( (37.iii)(38.ii)(38.iii) \)

(i) \( v = \langle d' \rangle \)  (ii) \( d' \in E_0^t \)  

by Lemma A.6  \( (40) \)

\( d' \gg d \)  

by inversion on \( (39.i) \),  \( (41) \)

using \( (40) \)

\( \text{run } \langle e' \rangle \xrightarrow{0^*} \text{run } \langle d' \rangle \xrightarrow{0} d' \gg d \)  

from \( (39.i)(40.i)(41) \)

[If \( t' \) small-steps]

(i) \( d = \text{run } d' \)  (ii) \( t' \xrightarrow{\ell} d' \)  (iii) \( d' \in E_{\ell}^t \)  

by inversion  \( (42) \)
\[ e' \gg t' \sim^\ell d' \quad \text{by (37.iii)(42.ii)} \quad (43) \]

\[(i) \ e' \sim^\ell t'' \gg d' \quad (ii) \ t'' \in E^\ell \quad \text{by IH} \]

\[\text{run } e' \sim^\ell \text{run } t'' \gg \text{run } d' \quad \text{immediately} \]

[PR-R]

\[(i) \ e = \text{run } \langle e' \rangle \quad (ii) \ e' \gg t \quad (iii) \ e', t \in E^0 \quad \text{by inversion} \quad (44)\]

\[e' \gg t \sim^\ell d \quad \text{by (44.ii) and premise}\]

\[\ell = 0 \quad \text{because (44.iii) but } t \sim^\ell d\]

\[(i) \ e' \sim^+_0 t' \gg d \quad (ii) \ t' \in E^\ell \quad \text{by IH}\]

\[\text{run } \langle e' \rangle \sim^0_0 e' \sim^+_0 t' \gg d \quad \text{immediately} \quad \square\]

### A.4 Proof Details for Anti-Equational Theory

This section fills in the details of proofs that showed the unsoundness of some equations in Section 3.3.

**Proof of Proposition 3.29.**

**Statement.** \( \forall C. \ 3L(C) \in \mathbb{N}. \lv e \geq L(C) \implies \lv C[e] = \lv e + \Delta C. \)

**Proof.** Induction on \(C.\)

[If \(C = \bullet\)] Take \(L(\bullet) \stackrel{\text{def}}{=} 0; \) then \(\lv C[e] = \lv e + L(\bullet).\)

For the remaining cases, I will take the existence of \(L(C')\) for granted, where \(C'\) names the immediate subcontext of \(C.\) This assumption is justified by IH. In each case, \(\lv e \geq L(C)\) is implicitly assumed once \(L(C)\) is defined.

[If \(C = \lambda x.C'\) or \(\text{run } C'\)] Take \(L(C) \stackrel{\text{def}}{=} L(C').\) Then \(\lv C[e] = \lv C'[e] = \lv e + L(C') = \lv e + L(C).\)
[If $C = t C'$ or $C' t$] Take $L(C) \overset{\text{def}}{=} \max(L(C'), lv t - \Delta C')$. Then $lv e + \Delta C' \geq L(C') + \Delta C' \geq lv t - \Delta C' + \Delta C' = lv t$, so $lv C[e] = \max(lv t, lv e + \Delta C') = lv e + \Delta C'$. Note that taking the maximum with $L(C')$ is necessary to justify IH.

[If $C = \langle C' \rangle$] Take $L(C) \overset{\text{def}}{=} \max(L(C'), 1 - \Delta C')$. Then $lv e + \Delta C' - 1 \geq 1 - \Delta C' + \Delta C' - 1 = 0$, so $lv C[t] = \max(lv C'[t] - 1, 0) = lv e + \Delta C' - 1 = lv e + \Delta \langle C' \rangle$.

Note that taking the maximum with $L(C')$ is necessary to justify IH.

[If $C = \tilde{C}'$] Take $L(C) \overset{\text{def}}{=} L(C')$. Then $lv C[e] = lv C'[t] + 1 = lv e + \Delta C' + 1 = lv e + \Delta (\tilde{C}')$. □

Lemma A.8. $lv e \leq \text{size}(e)$.

Proof. Straightforward induction on $e$. □

Proof of Lemma 3.31 (Context Domination).

Statement. $\ell > \text{size}(C) \implies C \in ECtx^{\ell,m}$.

Proof. Induction on $C$. There is a precondition $\ell' > \text{size}(C')$ for applying IH to the subcontext $C'$ to obtain $C' \in ECtx^{\ell',m}$. This precondition holds because IH is invoked with $\text{size}(C') \leq \text{size}(C) - 1$ and $\ell - 1 \leq \ell'$ in each case.

[If $C = \bullet$] Clearly $C \in ECtx^{\ell,\bullet}$, and $m \overset{\text{def}}{=} \ell$ satisfies $\Delta \bullet (m) = \ell$.

[If $C = C' e$]

\begin{align*}
C' e & \in ECtx^{\ell,m} \quad \text{by IH} \\
 lv e & \leq \text{size}(C) < \ell \quad \text{using Lemma A.8} \\
 e & \in E^\ell \quad \text{using Proposition 3.27 on (2)} \\
 C' e & \in ECtx^{\ell,m} \quad \text{by (1)(3)}
\end{align*}
[If $C = e C'$]

\[
C' \in ECtx^{\ell,m} \quad \text{by IH} \quad (4)
\]

\[
\text{lv } e \leq \text{size}(C') < \ell \quad \text{using Lemma A.8} \quad (5)
\]

\[
e \in E^{\ell-1} = V^\ell \quad \text{using Proposition 3.27 on (5)} \quad (6)
\]

\[
C' e \in ECtx^{\ell,m} \quad \text{by (4)(6)}
\]

[If $C = \langle C' \rangle$] IH gives $C' \in ECtx^{\ell-1,m}$, so $\langle C' \rangle \in ECtx^{\ell,m}$.

[If $C = \bar{C}'$] IH gives $C' \in ECtx^{\ell+1,m}$, so $\bar{C}' \in ECtx^{\ell,m}$.

[If $C = \text{run } C'$] IH gives $C' \in ECtx^{\ell,m}$, so $\text{run } C' \in ECtx^{\ell,m}$. \hfill \□

### A.5 Proof Details for Example 5.11

The proof in Example 5.11 relies on commutation of substitution, concluding $[a/x]e \approx [a/x]t$ from $\forall b. [b/x]e \approx [b/x]t$. This section proves Lemma 5.10, which justifies this inference.

**Lemma A.9.** $[e^0/x]a \in A$.

**Proof.** This lemma follows directly from Lemma 3.11 in CBN. In CBV, $a$ is of the form $\lambda y.t^0$ or $\langle t^0 \rangle$, and by BVC $[e^0/x](\lambda y.t^0) = \lambda y. [e^0/x]t^0$ or $[e^0/x]\langle t^0 \rangle = \langle [e^0/x]t^0 \rangle$, respectively. By Lemma 3.11 $[e^0/x]t^0 \in E^0$ so both of these forms are level-0 values. \hfill \□

**Lemma A.10.** Given a simultaneous substitution $\sigma : \text{Var} \xrightarrow{\text{fin}} A \overset{\text{def}}{=} [\overline{\alpha}/\overline{\alpha}_i]$ and a pair of expressions $e$ and $t$, there exists a sequential substitution $\sigma' \overset{\text{def}}{=} [\overline{c_j}/\overline{z_j}]^j [\overline{b_i}/\overline{x_i}]^i$ such that $\sigma' e \overset{\text{def}}{=} \sigma e$ and $\sigma' t \overset{\text{def}}{=} \sigma t$. Furthermore, the order in which the variables $\overline{x_i}$ are substituted for is arbitrary.
Proof. Observe that if either \( \forall i, j. x_i \not\in FV(a_j) \) or \( \forall i. a_i \in Var \) then a simultaneous substitution \([a_i/x_i]\) can be made sequential as \([a_i/x_i]\). Furthermore, in the former case the individual substitutions \([a_i/x_i]\) commute with each other, so their order is arbitrary. Thus using fresh variables \(z_i\),

\[
[a_i/x_i] = [x_i/z_i] \left( \left[ \frac{z_j^j/x_j^j}{a_i^i/x_i^i} \right] a_i^i/x_i^i \right) = [x_i/z_i] \left( \left[ \frac{z_j^j/x_j^j}{a_i^i/x_i^i} \right] a_i^i/z_i^i \right).
\]

But, unfortunately, \(x_i \not\in A\) in CBV, so the substitution \([x_i/z_i]\) does not have the signature \(Var \xrightarrow{fin} A\). By giving up syntactic equality between \(\sigma\) and \(\sigma'\), the new substitution can be made to have the required signature. I first choose an arbitrary \(v \in V_{cl}^0\) and substitute a stuck expression \(z_i v\) instead of just \(z_i\) for \(x_i\). Then I substitute \(\lambda_\_x_i\) for \(z_i\) to resolve this stuck application and contract it to \(x_i\) by \(\beta\) substitution.

\[
\sigma' = \left[\lambda_\_x_i/z_i\right] \left( \left[ \frac{z_j^j/v^j/x_j^j}{a_i^i/x_i^i} \right] a_i^i/x_i^i \right) = \left[\lambda_\_x_i/z_i\right] \left( \left[ \frac{z_j^j/v^j/x_j^j}{a_i^i/x_i^i} \right] a_i^i/z_i^i \right).
\]

Note that \(z_j^j v \not\in A\) is not a problem: this lemma only asserts \(\forall i. [z_j^j/v^j/x_j^j] a_i \in A\), which follows from Lemma A.9. Then for each \(i\),

\[
\sigma'x_i = \left[\lambda_\_x_i^j/z_j^j\right] \left[\frac{z_j^j/v^j/x_j^j}{a_i^i/x_i^i} \right] a_i = \left[\left(\lambda_\_x_i^j\right) v^j/x_j^j\right] a_i = \left[\frac{z_j^j/v^j/x_j^j}{a_i^i/x_i^i} \right] a_i = a_i = \sigma x_i.
\]

I omit the trivial induction argument that this equality extends to \(\sigma'e \equiv \sigma e\) and \(\sigma't \equiv \sigma t\).

\begin{flushright}
\square
\end{flushright}

Remark A.11. The only reason that I refer to a pair of expressions instead of one expression in Lemma A.10 is because I need fresh variables \(z_i\). If I request each \(z_i\) to be fresh for only \(\sigma\) and \(e\), then it might fail to be fresh for \(t\).

Proof of Lemma 5.10.

Statement. \(\forall a. [a/x]e \approx [a/x]t \implies \forall \sigma. [a/x]e \approx [a/x]t\).
Proof. Without loss of generality, $\sigma = \left[ b_i^{i \in I} / x_i^{i \in I} \right]$ where $I = \{1, 2, \ldots, \# \text{dom } \sigma \}$.

If $x \in \text{dom } \sigma$, then $[a/x]\sigma = \left[ a/b_i^{i \in I} / x_i^{i \in I} \right]$ and $\exists i. x = x_i$; if not, then $[a/x]\sigma = \left[ b_i^{i \in (0) \cup I} / x_i^{i \in (0) \cup I} \right]$ where $b_0 \overset{\text{def}}{=} a$ and $x_0 \overset{\text{def}}{=} x$. Either way, $[a/x]\sigma$ is equal to a single parallel substitution of the form $\sigma' \overset{\text{def}}{=} \left[ b_i^{i} / x_i^{i} \right]$ such that $\exists i. x_i = x$.

By Lemma A.10, there exists a sequential substitution $[a_i/x_i][a'/x]$ such that $\sigma'e \overset{\text{def}}{=} [a_i/x_i][a'/x]e$ and $\sigma't \overset{\text{def}}{=} [a_i/x_i][a'/x]t$. Note that the lemma explicitly states that I can require $x$ to be substituted first. Then

$$[a/x]\sigma e = \sigma'e \overset{\sim}{=} [a_i/x_i][a/x]e \approx [a_i/x_i][a'/x]t \overset{\sim}{=} \sigma't = [a/x]\sigma t.$$

A.6 Proof Details for Soundness and Completeness of Applicative Bisimulation

This section provides proof details pertaining to the soundness and completeness of indexed applicative bisimulation.

Proof of Lemma 5.30.

Statement. If $R_X$ is a family of relations, then

(i) $e[R]_{X} t \implies \forall \sigma : X \mid \text{Var} \overset{\text{fin}}{\rightarrow} A. \sigma e[R]_{X \setminus \text{dom } \sigma} \sigma t$ and

(ii) $\forall Y \subseteq X. ((\forall \sigma : Y \mid \text{Var} \overset{\text{fin}}{\rightarrow} A. \sigma e[R]_{X \setminus \text{dom } \sigma} \sigma t) \implies e[R]_{X} t)$.

In particular, $e \lesssim_X t \iff \forall \sigma : X \mid \text{Var} \overset{\text{fin}}{\rightarrow} A. \sigma e \lesssim_{\emptyset} \sigma t$.

Proof.

(i) Suppose $e[R]_{X} t$ and let $\sigma : \text{Var} \overset{\text{fin}}{\rightarrow} A$ be given. Then for any $\sigma' : (X \setminus \text{dom } \sigma) \mid \text{Var} \overset{\text{fin}}{\rightarrow} A$, the composition of the substitutions satisfies $\sigma' \sigma : X \mid \text{Var} \overset{\text{fin}}{\rightarrow} A$, where $(\sigma' \sigma)e \overset{\text{def}}{=} \sigma'(\sigma e)$. Thus by assumption $\sigma'(\sigma e) \Downarrow^t v \implies (\sigma'(\sigma t) \Downarrow^t u \land v \{R_Y\}^t u)$. Therefore, $(\sigma e)R_{X \setminus \text{dom } \sigma}(\sigma t)$.
(ii) Suppose \( \forall \sigma : Y \mid Var \xrightarrow{\text{fin}} A. (\sigma e) [R] \setminus \text{dom} \sigma (\sigma t) \) for some \( Y \) and let any \( \sigma' : X \mid Var \xrightarrow{\text{fin}} A \) be given. By the assumption \( Y \subseteq X \subseteq \text{dom} \sigma' \) and Lemma A.10, \( \sigma' \) can be decomposed as \( \sigma' = \sigma'' \sigma \) for some \( \sigma : Y \mid Var \xrightarrow{\text{fin}} A \) and \( \sigma'' : X \setminus \text{dom} \sigma \xrightarrow{\text{fin}} A \) such that \( \sigma''(\sigma e) \vdash \sigma' e \) and \( \sigma''(\sigma t) \vdash \sigma' t \). Then by assumption \\
(\sigma e) [R] \setminus \text{dom} \sigma (\sigma t), \text{ so } \sigma''(\sigma e) \Downarrow^\ell v \Rightarrow ((\sigma''(\sigma t) \Downarrow^\ell u \land v \{R\}^\ell u).

The statement about \((\subseteq_\emptyset)\) follows by taking \( R = (\subseteq) \) and \( Y = X \).

**Proof of Lemma 5.31.**

**Statement.** \( e \lessapprox_X t \wedge a \lessapprox_X \{x\} b \implies [a/x]e \lessapprox_X \{b/x\} t \).

**Proof.** Induction on \( e \) with case analysis on \( e \).

[If \( e = x \)] This case was proved in the main text.

[If \( e = \tau \overline{e_i} \neq \lambda x.e^0 \)]

\[
\begin{align*}
(1) & \quad \overline{e_i} \lessapprox_X d_i \quad \text{(ii) } \tau \overline{d_i} \lessapprox_X t \quad \text{by assumption} \\
(2) & \quad [a/x]e_i \lessapprox_X [b/x]d_i \quad \text{by IH on (1.i)} \\
(3) & \quad \begin{array}{l}
\tau [a/x]e_i = [a/x](\tau \overline{e_i}) \\
\tau [b/x]d_i = [b/x](\tau \overline{d_i})
\end{array} \quad \text{using BVC} \\
(4) & \quad [a/x](\tau \overline{e_i}) \lessapprox_X [b/x](\tau \overline{d_i}) \quad \text{by (2)(3) and Proposition 5.28 (ii)} \\
(5) & \quad [b/x](\tau \overline{d_i}) \lessapprox_X \{x\} [b/x] t \quad \text{by (1.ii) using Lemma 5.30} \\
(6) & \quad [a/x](\tau \overline{e_i}) \lessapprox_X \text{ by (4)(5) using Proposition 5.28 (iii)}
\end{align*}
\]

[If \( e = \lambda y.e' \) where \( e' \in E^0 \)]

\[
\begin{align*}
(1) & \quad e' \lessapprox_Y d' \quad \text{(ii) } \lambda y.d' \lessapprox_X t \quad \text{(iii) } Y \setminus \{y\} = X \quad \text{by inversion} \\
(2) & \quad [a/x]e' \lessapprox_{Y \setminus \{x\}} [b/x]d' \quad \text{by IH on (6.i)}
\end{align*}
\]
\[ \lambda_y. [a/x] e' = [a/x] \lambda y. e' \]
\[ \lambda_y. [b/x] d' = [b/x] \lambda y. d' \]
\[ [a/x] \lambda y. e' \in E^0 \]
\[ [a/x] \lambda y. e' \preceq_{Y \setminus \{x,y\}} [b/x] \lambda y. d' \]
\[ [a/x] \lambda y. e' \preceq_{X \setminus \{x\}} [b/x] \lambda y. d' \]
\[ [b/x] \lambda y. d' \preceq_{X \setminus \{x\}} [b/x] t \]
\[ [a/x] \lambda y. e' \preceq_{X \setminus \{x\}} [b/x] t \]

using BVC \hspace{1cm} (8)
by Lemma 3.11 and \( e' \in E^0 \) \hspace{1cm} (9)
by \( (7)(8)(9) \) \hspace{1cm} (10)
using (6.ii) \hspace{1cm} (11)
by \( (6.ii) \) and Lemma 5.30

\[ \lambda y. [a/x] e' = [a/x] \lambda y. e' \]

**Lemma A.12.** \( v \{ \widetilde{\leq}_X \}^\ell w \{ \leq_X \}^\ell u \implies v \{ \widetilde{\leq}_X \}^\ell u. \)

**Proof.** Easily confirmed by inspecting Definition 5.24 using Proposition 5.28 (iii).

**Proof of Lemma 5.34.**

Statement. \( e \preceq_X t \implies e[\preceq_X]t \)

**Proof.** Fix a \( \sigma \) and an \( \ell \), and assume \( \sigma e, \sigma t \in E^\ell \land \sigma e \sim^n \sigma v \). I will show \( \sigma t \downarrow^\ell u \land v \{ \widetilde{\leq}_X \}^\ell u \) by lexicographic induction on \( (n,e) \) with case analysis on the form of \( e \).

[If \( e = x \)]

\[ x \preceq_X t \quad \text{by inversion} \hspace{1cm} (1) \]
\[ \sigma x \preceq_{X \setminus \{x\}} \sigma t \quad \text{by Lemma 5.30} \hspace{1cm} (2) \]
\[ (i) \sigma t \downarrow^\ell u \quad (ii) v \{ \widetilde{\leq}_X \}^\ell u \quad \text{by (2) and } \{ \widetilde{\leq}_X \} = \{ \leq_X \} \hspace{1cm} (3) \]
\[ (i) \sigma t \downarrow^\ell u \quad (ii) v \{ \widetilde{\leq}_X \}^\ell u \quad \text{by Lemma 5.30} \hspace{1cm} (4) \]

For the remaining cases, \( e \) has the form \( e = \tau \overline{e_i} \) and

\[ (i) \overline{e_i} \preceq_X \overline{d_i} \quad (ii) \tau \overline{d_i} \preceq_X t \quad \text{by inversion} \hspace{1cm} (5) \]
Then it suffices to show $\sigma(\tau \overline{d_i}) \not\equiv^\ell w$ and $v\{\overline{\overline{\leq_X}}\}^\ell w$ because in that case

$$v\{\overline{\overline{\leq_X}}\}^\ell u \qquad \text{by Lemma A.12 using (5.ii)(6.ii)}$$

[If $e = e_1 e_2$] Noting that $\bullet \sigma e_2 \in ECtx^\ell,\ell$ and $\forall v_1 \in V^\ell. \forall v_1 \bullet \in ECtx^\ell,\ell$,

(i) $\sigma e_i \not\equiv_{\ell}^n v_i$ ($i = 1, 2$) \hspace{1cm} (ii) $v_1, v_2 \in V^\ell$ \hspace{1cm} using Lemma 3.6

(iii) $v_1 v_2 \not\equiv_{\ell}^{(n-n_1-n_2)} v$ \hspace{1cm} (iv) $n_1, n_2 \leq n$

Now split cases on $\ell$.

[If $\ell = 0$]

(i) $v_1 = \lambda x.e'$ \hspace{1cm} (ii) $e' \in E^0$

(iii) $v_1 v_2 \not\equiv_{0}^{[v_2/x]}e'$ \hspace{1cm} by inversion on (7.iii)

(iv) $[v_2/x]e' \not\equiv_{0}^{(n-n_1-n_2-1)} v$

[If $\ell > 0$]

$(v_1 v_2) \in V^\ell$ \hspace{1cm} by (7.ii)

$(w_1 w_2) \in V^\ell$ \hspace{1cm} by (8.ii)

$v_i \not\equiv_{0} w_i$ ($i = 1, 2$) \hspace{1cm} by (8.iii)

$v_1 v_2 \not\equiv_{0} w_1 w_2$ \hspace{1cm} by Proposition 5.28 (ii)

$v_1 v_2 \{\overline{\overline{\leq_X}}\}^\ell w_1 w_2$ \hspace{1cm} by definition
For the remaining cases $e$ has the form $\tau\ e'$ and (5) instantiates as

\[(i) \ e' \overset{\prec}{\sim} \ d' \quad (ii) \ \sigma\ d' \overset{\prec}{\sim} \ t. \quad (11)\]

Once again, I only need to show $\sigma(\tau\ d' \Downarrow^\ell w \land v \{\overset{\prec}{\sim} \}^\ell w)$.

[If $e = \lambda x.e'$]

[If $\ell = 0$] Let $d \overset{\text{def}}{=} \lambda x.d'$. Then $\sigma e, \sigma t \in V^0$ so these expressions terminate to themselves. By (11.i) and Lemma 5.31, $\sigma e \overset{\prec}{\sim} \emptyset \sigma d$.

[If $\ell > 0$] By BVC, $\sigma e = \lambda x.\sigma e'$ and $\sigma d = \lambda x.\sigma d'$. Thus, noting that $\lambda x.\bullet \in ECtx^{\ell,\ell}$,

\[(i) \ \sigma e' \overset{\sim}{\sim}^{\ell+1} v' \quad (ii) \ v = \lambda x.v' \quad (iii) \ v \in V^{\ell+1} \quad \text{using Lemma 3.6} \quad (12)\]
\[(i) \ \sigma d' \Downarrow^{\ell+1} w' \quad (ii) \ v' \{\overset{\prec}{\sim} \}^{\ell+1} w' \quad (iii) \ w' \in V^{\ell+1} \quad \text{by IH on (12.i)} \quad (13)\]
\[
v' \overset{\prec}{\sim} \emptyset w' \quad \text{from (13.ii)}
\]
\[
\lambda x.v' \overset{\prec}{\sim} \emptyset \lambda x.w' \\
\lambda x.v' \{\overset{\prec}{\sim} \}^{\ell} \lambda x.w' \quad \text{by Proposition 5.28 (ii)}
\]

[If $e = \langle e' \rangle$] Noting that $\langle \bullet \rangle \in ECtx^{\ell,\ell+1}$,

\[(i) \ \sigma e' \overset{\sim}{\sim}^{\ell+1} v' \quad (ii) \ v' \in V^{\ell+1} \quad (iii) \ n' \leq n \quad \text{by Lemma 3.6} \quad (14)\]
\[(i) \ \sigma d' \Downarrow^{\ell+1} w' \quad (ii) \ v' \{\overset{\prec}{\sim} \}^{\ell+1} w' \quad (iii) \ w' \in V^{\ell+1} \quad \text{by IH on (14.i)} \quad (15)\]
\[
v' \overset{\prec}{\sim} \emptyset w' \quad \text{by (15.ii)} \quad (16)\]
\[
\langle v' \rangle \overset{\prec}{\sim} \emptyset \langle w' \rangle \quad \text{by Proposition 5.28 (ii)} \quad (17)\]
\[
\langle v' \rangle \{\overset{\prec}{\sim} \}^{\ell} \langle w' \rangle \quad \text{by (16) if } \ell = 0; \text{ or } \quad (18)\]

[If $e = \tilde{e}'$]

$\ell > 0$ because $\tilde{e}' \in E^{\ell}$
\[ \neg \bullet \in ECtx^{\ell,\ell-1} \]

(i) \( \sigma e' \sim_{\ell-1}^{n'} v' \) (ii) \( v' \in V^{\ell-1} \) by (18) (19)

(i) \( \sigma d' \Downarrow_{\ell-1}^{w'} w' \) (ii) \( v' \{ \frac{X}{\sim_{X}} \}_{\ell-1}^{\ell-1} w' \)

(iii) \( w' \in V^{\ell-1} \) by IH on (20.i) (21)

Split cases on \( \ell \).

[If \( \ell = 1 \)] Clearly \( \neg v' \not\in E^0 = V^1 \), so \( \neg v' \) must small-step at level 1.

(i) \( v' = \langle e'' \rangle \) (ii) \( e'' \in E^0 = V^1 \) by inversion, using (20.ii) (22)

(i) \( w' = \langle d'' \rangle \) (ii) \( d'' \in E^0 = V^1 \) by (21.ii)(22.i)

\( \neg d' \sim_1 - \langle d'' \rangle \Downarrow^1 d'' \) immediately

[If \( \ell > 1 \)]

\( v', w' \in V^{\ell-1} = E^{\ell-2} \)

\( \neg v', \neg w' \in V^{\ell-1} = E^{\ell-2} \) immediately

\( v' \overset{\sim}{\sim}_0 w' \) because (21.ii) and \( \ell - 1 > 0 \)

\( \neg v' \overset{\sim}{\sim}_0 \neg w' \) by Proposition 5.28 (ii)

\( \neg v' \{ \frac{X}{\sim_X} \}_{\ell}^{\ell} \neg w' \) immediately

[If \( e = \text{run } e' \)] Noting that \( \text{run } \bullet \in ECtx^{\ell,\ell} \),

(i) \( \sigma e' \sim_{\ell}^{n'} v' \) (ii) \( v \in V^\ell \) (iii) \( n' \leq n \) by Lemma 3.6 (23)

(i) \( \sigma d' \Downarrow_{\ell}^{w'} w' \) (ii) \( v' \{ \frac{X}{\sim_X} \}_{\ell}^{\ell} w' \) (iii) \( w' \in V^\ell \) by IH on (23.i) (24)

Split cases on \( \ell \).

[If \( \ell = 0 \)] Since \( \text{run } v' \in V^0 \), the \( \text{run } v' \) must small-step.

(i) \( v' = \langle e'' \rangle \) (ii) \( e'' \sim_{0}^{n''} v \)

(iii) \( e'' \in E^0 \) (iv) \( n'' \leq n' \) by inversion (25)
(i) \( w' = \langle d'' \rangle \)  
(ii) \( e'' \overset{\sim}{\lesssim}_0 d'' \)  
(iii) \( d'' \in E^0 \) by (24.ii)(24.iii)(25.ii)  
(26)

(i) \( d'' \not\lessim_0 w \)  
(ii) \( v \{ \overset{\sim}{\lesssim}_X \}^0 w \) by IH on (25.ii)  
(27)

(i) \( \sigma(\text{run } \langle d' \rangle) \sim^* \text{run } \langle d'' \rangle \sim d'' \not\lessim_0 w \) by (24)

[If \( \ell > 0 \)]

\[ \begin{align*}
\text{run } v', \text{run } w' \in V^\ell & \quad \text{by (23.ii)(24.iii)} \\
v' \overset{\sim}{\lesssim}_0 w' & \quad \text{by (24.ii)} \\
\text{run } v' \overset{\sim}{\lesssim}_0 \text{run } w' & \quad \text{by Proposition 5.28 (ii)} \\
\text{run } v' \{ \overset{\sim}{\lesssim}_X \}^\ell \text{run } w' & \quad \text{immediately}
\end{align*} \]
Appendix B

Summary of Notations

The following figures summarize the mathematical notations used in this document. Table B.1 lists the syntax for relations and annotations. Entries are sorted in order of appearance in the main text.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>at level $\ell$ (constraint)</td>
<td>$e^\ell$, $v^\ell$, etc.</td>
</tr>
<tr>
<td>hole</td>
<td>$\bullet$</td>
</tr>
<tr>
<td>term size</td>
<td>$\text{size}(e)$</td>
</tr>
<tr>
<td>context size</td>
<td>$\text{size}(C)$</td>
</tr>
<tr>
<td>syntactic equality</td>
<td>$e = t$</td>
</tr>
<tr>
<td>primitive reduction</td>
<td>$e \xrightarrow{m}^\text{prim} t$</td>
</tr>
<tr>
<td>small-step</td>
<td>$e \xrightarrow{\ell} t$</td>
</tr>
<tr>
<td>observational equivalence</td>
<td>$e \approx t$</td>
</tr>
<tr>
<td>termination</td>
<td>$e \downarrow^\ell$</td>
</tr>
<tr>
<td>divergence</td>
<td>$e \uparrow^\ell$</td>
</tr>
<tr>
<td>reduction</td>
<td>$e \rightarrow t$</td>
</tr>
<tr>
<td>provable equality</td>
<td>$e \simeq t$</td>
</tr>
<tr>
<td>free variables</td>
<td>$\text{FV}(e)$</td>
</tr>
<tr>
<td>reflexive-transitive closure</td>
<td>$R^*$</td>
</tr>
<tr>
<td>transitive closure</td>
<td>$R^+$</td>
</tr>
<tr>
<td>finite iteration</td>
<td>$R^n$</td>
</tr>
<tr>
<td>parallel reduction</td>
<td>$e \nRightarrow t$</td>
</tr>
<tr>
<td>erasure</td>
<td>$|e|$</td>
</tr>
<tr>
<td>CBN variant</td>
<td>$eR^\text{m}_n t$</td>
</tr>
<tr>
<td>CBV variant</td>
<td>$eR^\text{v}_n t$</td>
</tr>
<tr>
<td>careful reduction</td>
<td>$e \rightarrow^\text{vc} t$</td>
</tr>
<tr>
<td>careful equality</td>
<td>$e \simeq^\text{vc} t$</td>
</tr>
<tr>
<td>observational order</td>
<td>$e \preccurlyeq t$</td>
</tr>
<tr>
<td>applicative simulation</td>
<td>$e \lesssim_X t$</td>
</tr>
<tr>
<td>applicative bisimulation</td>
<td>$e \sim_X t$</td>
</tr>
<tr>
<td>precongruence candidate</td>
<td>$e \lesssim_X t$</td>
</tr>
</tbody>
</table>

Table B.1 : Summary of notations.