JFP **26**, e22, 53 pages, 2016.

doi:10.1017/S0956796816000253

*Reasoning about multi-stage programs**

JUN INOUE

National Institute of Advanced Industrial Science and Technology, Ikeda, Osaka, Japan
(*e-mails:* jun.inoue@aist.go.jp)

WALID TAHA

Halmstad University, Halmstad, Halland, Sweden
(*e-mails:* walid.taha@hh.se)

Abstract

We settle three basic questions that naturally arise when verifying code generators written in multi-stage functional programming languages. First, does adding staging to a language compromise any equalities that hold in the base language? Unfortunately it does, and more care is needed to reason about terms with free variables. Second, staging annotations, as the name "annotations" suggests, are often thought to be orthogonal to the behavior of a program, but when is this formally guaranteed to be true? We give termination conditions that characterize when this guarantee holds. Finally, do multi-stage languages satisfy useful, standard extensional properties, for example, that functions agreeing on all arguments are equivalent? We provide a sound and complete notion of applicative bisimulation, which establishes such properties or, in principle, any valid program equivalence. These results yield important insights into staging and allow us to prove the correctness of quite complicated multi-stage programs.

1 Introduction

Multi-stage programming (MSP) allows programmers to write generic code without sacrificing performance. Programmers can write code generators that are themselves generic but generate specialized, efficient code (Brady & Hammond, 2006; Cohen *et al.*, 2006; Herrmann & Langhammer, 2006; Carette *et al.*, 2009; Carette & Kiselyov, 2011). However, few formal studies have considered verifying generators written with MSP, and MSP research has predominantly focused on applications that confirm performance benefits and on type systems (Taha & Nielsen, 2003; Kim *et al.*, 2006; Yuse & Igarashi, 2006; Tsukada & Igarashi, 2010; Westbrook *et al.*, 2010; Kameyama *et al.*, 2011).

This gap in the literature is a significant shortcoming, as ensuring the correctness of code generators can be challenging. Fixing errors in the generator often entails

* This work was supported by NSF CCF 0747431 award entitled "Multi-stage programming for object-oriented languages", NSF CSR/EHS 0720857 award entitled "Building physically safe embedded systems", NSF CPS 1136099 award entitled "A CPS Approach to Robot Design", and Halmstad University.

figuring out which pieces of code came from where in the generator and why. This task can be time-consuming even with tool support, because programmers must understand the unfamiliar body of code produced by the generator. This problem is exacerbated when the generated code is heavily optimized and can change drastically as the generator changes, or if many variants of the code must be generated. Programmers would be better served by being able to minimize inspection of the generated code, for then they would only have to deal with familiar code. Moreover, a generator may produce problematic code only for certain inputs while working fine on other inputs. Verifying the generator, as opposed to individual generated instances, allows us to verify the entire family of programs that the generator can produce, giving greater payoff for the verification effort.

To address this shortcoming, we advocate in this paper an approach to verifying code generators that minimizes the need to contemplate the generated code. The idea is to find conditions under which the constructs related to code generation are semantics-preserving and can be safely ignored. The power function gives a good, concise example for demonstrating this approach, presented here in MetaOCaml syntax.[1]

```
let rec power n x =
  if n = 1 then x
  else x * power (n-1) x
let rec genpow n x =
  if n = 1 then x
  else <~x * ~(genpow (n-1) x)>
let stpow n = ! <fun z -> ~(genpow n <z>)>
```

This code defines a function named `power` which maps x and n to x^n. The `power` function subsumes all functions of the form `fun x -> x*x*...*x`, but every time it is called, it wastes time on recursive calls and conditional branches. Staging annotations in `genpow` eliminate this overhead by resolving the branches and unrolling the recursion. Brackets `<`e`>` delay an expression e. An escape `~`e must occur within brackets and causes e to be evaluated without delay, locally undoing the effect of surrounding brackets. The e should return a value of the form `<`e'`>`, and e' replaces `~`e. Run `!`e compiles and runs the code generated by e. These annotations in MetaOCaml are hygienic, i.e., preserve static scoping (Dybvig, 1992), but are otherwise like LISP's quasiquote, unquote, and eval (Muller, 1992). The `genpow` function uses these constructs to generate, for any concrete `n`, a compiled function that performs only multiplication.

[1] We have taken the liberty to omit the periods that MetaOCaml demands around staging constructs. Otherwise, code listings faithfully render the syntax accepted by MetaOCaml.

For example, the evaluation of `stpow 2` proceeds as follows:

```
   stpow 2
⇝+                                                    (1)
   ! <fun z -> ~(genpow 2 <z>)>
⇝+                                                    (2)
   ! <fun z -> ~(<z * z>)>
⇝+                                                    (3)
   ! <fun z -> z * z>
⇝+                                                    (4)
   fun z -> z * z
```

where $\rightsquigarrow^+$ stands for one or more steps of evaluation (formally defined in Section 2). Step (1) is just unfolding the definition of `stpow`. Step (2) evaluates the escaped part (`genpow 2 <z>`) of the code being generated. Note that this step evaluates an open term; escape is forcing evaluation to occur under the binder `fun z`, which is a distinguishing feature of MSP. Step (3) splices the generated code into the surrounding context to create a bigger code value. Finally, at step (4), `!` compiles and executes the generated code, yielding a closure. This closure, when called, performs nothing but multiplication.

This example is typical of MSP usage, where a staged program `stpow` is meant as a drop-in replacement for the unstaged program `power`. Given `stpow`, we can reconstruct the unstaged program `power` by erasing staging annotations—we say that `power` is the *erasure* of `stpow`. In light of the similarity of these programs, if we are to verify `stpow`, we naturally expect `stpow` $\approx$ `power` to hold for some suitable program equivalence ($\approx$) and hope to get away with proving that `power` satisfies whatever specifications it has, *in lieu* of `stpow`. Then, `power` can be analyzed straightforwardly by conventional reasoning techniques designed for single-stage programs. But three key concerns must be addressed before we can apply this strategy with confidence:

Conservativity. Do all reasoning principles valid in a single-stage language carry over to its multi-stage extension?

Conditions for Sound Erasure. In the power example, staging seems to preserve semantics, but clearly this is not always the case: If Ω is non-terminating, then $\langle\Omega\rangle \not\approx \Omega$ for any sensible ($\approx$). When do we know that erasing annotations preserves semantics?

Extensional Reasoning. How, in general, do we prove equivalences of the form $e \approx t$? It is known that hygienic, purely functional MSP satisfies intensional equalities like β (Taha, 1999), but those equalities are too weak to prove such properties as extensionality (i.e., functions agreeing on all inputs are equivalent). Extensional facts are indispensable for reasoning about functions, like `stpow` and `power`.

This paper settles these questions for the untyped, purely functional case with hygiene. We work without types to avoid committing to the particulars of any specific type system, since there are multiple useful type systems for MSP (Taha

& Nielsen, 2003; Tsukada & Igarashi, 2010; Westbrook *et al*., 2010; Kameyama *et al*., 2015). It also ensures that our results apply to dynamically typed languages (Dybvig, 1992), where hygienic code generation is just as useful as in statically typed languages.

Hygiene, or the absence of inadvertent variable capture that it ensures, is a widely accepted safety feature that ensures many of the nice theoretical properties of MSP, which helps to reason about programs, and which we exploit in this study. This is an important point of difference from Choi *et al*. (2011). They also advocate another approach that eliminates code generation but in a semantics that has variable capture and delegates capture avoidance to an explicit "gensym" construct. Their approach has different trade-offs working in different settings, so our development and theirs fill complementary roles.

We believe imperative hygienic MSP is not yet ready for an investigation like this. Types are essential for having a sane operational semantics without scope extrusion (Kameyama *et al*., 2011), but there is no decisive solution to this problem, and the jury is still out on many of the trade-offs. The foundations for imperative hygienic MSP have not matured to the level of the functional theory that we build upon here.

Tagless final encodings (Carette *et al*., 2009), and the lightweight modular staging framework (Rompf & Odersky, 2012) inspired by that technique, give a different approach to metaprogramming than MetaOCaml-style MSP. They offer data types that not only represent code, like bracketed expressions do in MetaOCaml, but can also be interpreted by any semantics of the user's choosing. The semantics may evaluate the code, print the code, or perform a post-generation-pass optimization and emit some intermediate representation. These frameworks are not limited to staging (separating a program into multiple execution phases, or stages) but rather support general-purpose metaprogramming (writing programs that manipulate other programs in arbitrary ways). Reasoning in those frameworks depends on the semantics given to the object code, so it is beyond the scope of this paper. However, our approach may still be relevant when the machinery is used specifically for staging.

1.1 Contributions

We extend previous work on the call-by-name (CBN) multi-stage λ calculus, λ^U (Taha, 1999), to cover call-by-value (CBV) as well (Section 2). In this calculus, we show the following results.

Unsoundness of Reasoning Under Substitutions. Unfortunately, the answer to the conservativity question is "no". Because λ^U can express open-term manipulation (see `genpow` above), equivalences proved under closing substitutions are not always valid without substitution, for such a proof implicitly assumes that only closed terms are manipulated at runtime. We illustrate how this pathology occurs using the surprising fact $(\lambda\_.0)\ x \not\approx 0$, and explain what can be done about it (Section 3). The rest of the paper will show that λ^U nonetheless conserves a wealth of useful reasoning principles. Many familiar proof rules and techniques carry over from the

plain λ calculus so that a lot can be achieved, despite the fact that we can no longer focus our attention exclusively to closed instances of terms.

Conditions for Sound Erasure. We show that reductions of a staged term are simulated by equational rewrites of the term's erasure. This gives simple termination conditions that guarantee erasure to be semantics-preserving (Section 4). Considering CBV in isolation turns out to be unsatisfactory, and borrowing CBN facts is essential in establishing the termination conditions for CBV. Intuitively, this happens because annotations change the evaluation strategy, and the CBN equational theory subsumes reductions in all other strategies whereas the CBV theory does not.

Soundness of Extensional Properties. We give a sound and complete notion of applicative bisimulation (Abramsky, 1990; Gordon, 1999) for λ^U. Bisimulation gives a general extensional proof principle that, in particular, proves extensionality of λ abstractions. It also justifies reasoning under substitutions in some cases, limiting the impact of the non-conservativity result (Section 5).

To demonstrate the wide applicability of our methods, we present substantial case studies proving the correctness of non-trivial generators (Section 6). In Section 6.1, we verify the LCS algorithm, which is staged into a sophisticated code generator that couples `let`-insertion with continuation-passing style and monadic memoization using the techniques of Swadi *et al.* (2006). These techniques make an exact description of the generated code hard to pin down, but our result on erasure makes such details irrelevant. We also verify a generator for `fold` (Section 6.2), which demonstrates that higher order generators are also amenable to our verification methodology.

Throughout the paper, we emphasize the general insights about MSP that can be gleaned from our results. In particular, we find that CBN is better behaved than CBV, as metaprogrammers who have experience with MSP in both settings may have already come to realize. The shortcomings of CBV stem largely from premature evaluation of subexpressions that may diverge, and a large part of our effort consists in building tools to reason in the face of that obstacle. Though we do stress the applicability to verification, we strive to establish a deep, general understanding of staging, and let the tools for verification fall out as natural byproducts. We demonstrate those tools along the way, using the power function as a running example.

This paper is a summary and an extension of the first author's doctoral thesis (Inoue, 2012), which was previously published at a conference (Inoue & Taha, 2012). This paper incorporates materials from the thesis that were relegated to a technical report in the conference version due to space limitations, as well as some new results:

- A detailed discussion of why certain generalizations to the equational theory are unsound (Section 2.3). Together with the issue of reasoning under substitutions (Section 3), this discussion gives a thorough understanding of where the boundary lies between equalities that hold in a multi-stage language and those that don't.

$$
\begin{array}{ll}
\textit{Levels} \quad \ell, m \in \mathbb{N} & \textit{Variables} \quad x, y \in \mathit{Var} \qquad \textit{Constants} \quad c, d \in \mathit{Const} \\
\textit{Expressions} & e, t, s \in E ::= c \mid x \mid \lambda x.e \mid e\ e \mid \langle e \rangle \mid \tilde{}e \mid !e \\
\textit{Contexts} & C \in Ctx ::= \bullet \mid \lambda x.C \mid C\ e \mid e\ C \mid \langle C \rangle \mid \tilde{}C \mid !C \\
\textit{Exact Level} & \mathrm{lv} : E \to \mathbb{N} \text{ where} \\
 & \quad \mathrm{lv}\, x \stackrel{\text{def}}{=} 0 \quad \mathrm{lv}\, c \stackrel{\text{def}}{=} 0 \quad \mathrm{lv}(e_1\ e_2) \stackrel{\text{def}}{=} \max(\mathrm{lv}\, e_1, \mathrm{lv}\, e_2) \quad \mathrm{lv}(\tilde{}e) \stackrel{\text{def}}{=} \mathrm{lv}\, e + 1 \\
 & \quad \mathrm{lv}(\lambda x.e) \stackrel{\text{def}}{=} \mathrm{lv}\, e \qquad \mathrm{lv}\langle e \rangle \stackrel{\text{def}}{=} \max(\mathrm{lv}\, e - 1, 0) \quad \mathrm{lv}(!e) \stackrel{\text{def}}{=} \mathrm{lv}\, e \\
\textit{Programs} & p \in \mathit{Prog} \stackrel{\text{def}}{=} \{e : \mathrm{lv}\, e = 0 \wedge \mathrm{FV}(e) = \varnothing\} \\
\textit{Stratification} & e^\ell, t^\ell, s^\ell \in E^\ell \stackrel{\text{def}}{=} \{e : \mathrm{lv}\, e \leq \ell\} \\
\textit{Values} & u^0, v^0, w^0 \in V^0 ::= c \mid \lambda x.e^0 \mid \langle e^0 \rangle \\
 & u^{\ell+1}, v^{\ell+1}, w^{\ell+1} \in V^{\ell+1} ::= e^\ell
\end{array}
$$

Fig. 1. Syntax of λ^U, parameterized in a set of constants *Const*.

- A significantly improved, nuanced definition of careful equalities (Section 4.5), used for proofs in CBV. In the conference paper, this technique was not developed enough to be a serious contender to the normalization technique presented in Section 4.3, but we have succeeded in reformulating it to have a clear advantage in analyzing higher order generators. This material is new.
- Proofs of soundness and completeness of applicative bisimulation (Appendix A.2).
- The verification example of a higher order generator (Section 6.2). This material is new.

This paper supersedes the conference version. It gives more proof details and explanations than the conference version, but at a level that keeps the flow and should be easy to follow. The thesis writes out all proofs in meticulous detail in an appendix, so readers interested in working out, checking, or mechanizing the proofs may find the thesis to be a valuable complement. Reading the thesis is not necessary to understand this paper, however.

2 The λ^U calculus: Syntax, semantics, and equational theory

This section introduces λ^U, a simple but expressive calculus that models all possible uses of brackets, escape, and run in MetaOCaml's purely functional core, sans types. The syntax and operational semantics of λ^U for both CBN and CBV are minor extensions of previous work (Taha, 1999) to allow arbitrary constants. The CBN equational theory is more or less as in Taha (1999), but the CBV equational theory is new.

Notation. A set S may be marked as CBV ($S_{\mathbf{v}}$) or CBN ($S_{\mathbf{n}}$) if its definition varies by evaluation strategy. The subscript is dropped in assertions and definitions if they apply to both evaluation strategies or if clear from context. Syntactic equality (α equivalence) is written ($\equiv$). The set of free variables in e is written $\mathrm{FV}(e)$.

2.1 Syntax and operational semantics

Figure 1 shows the syntax of λ^U. The set of terms is that of the plain λ calculus extended with constants and the three staging primitives brackets, escape, and run. A

context C is an incomplete term containing exactly one hole $\bullet$ in place of a subterm. The result of plugging, or replacing, the hole by e is written $C[e]$, where binders in the context C capture free variables in e. The exact level of a term is its nesting depth of escapes, where a pair of brackets cancels one level of escape, provided the brackets enclose the escape (and not the other way around). A program is a closed term with exact level 0.

Levels are used to encode the following rules for nesting brackets and escapes: (a) a term is delayed iff more brackets surround it than do escapes, and (b) in a program every escape must occur in a delayed region. For example, in the following terms, e_1 and e_2 are delayed while e_3, e_4, and e_5 are not. The term containing e_6 is not a valid program (though it can be a subterm of a valid program), and it makes no sense to ask whether e_6 should be delayed.

$$\langle e_1 \rangle \qquad \langle\langle \tilde{} e_2 \rangle\rangle \qquad \langle \tilde{} e_3 \rangle \qquad \langle\langle \tilde{}\tilde{} e_4 \rangle\rangle \qquad \langle \tilde{} \langle \tilde{} e_5 \rangle\rangle \qquad \langle \tilde{}\tilde{} e_6 \rangle$$

A term like $\tilde{}\tilde{} e_6$ is not self-contained in the sense that it cannot appear just anywhere in a program, but must appear inside at least two pairs of brackets. Otherwise, the inner $\tilde{}$ would fall within a non-delayed part of the program term and have no delay to cancel. A term's exact level is the minimum number of brackets it must appear within in a valid program.

We say that a context C is a *program context* for e iff $C[e]$ is a program. Because $\mathrm{lv}\, e \leqslant \mathrm{lv}\, t$ implies that any program context for t is a program context for e as well, upper bounds for a term's level are usually more useful than the exact level. Thus, we often say "e has level ℓ" or "e is a level-ℓ term", written $e \in E^{\ell}$, to mean $\mathrm{lv}\, e \leqslant \ell$. We say "$e$ has *exact* level ℓ", explicitly using the keyword "exact", when we mean $\mathrm{lv}\, e = \ell$.

A level-0 value (i.e., a value in a non-delayed region) is a constant, an abstraction, or a code with no undelayed region. A level-0 value of the form $\langle e^0 \rangle$ is called a *code value*. At level $\ell > 0$ (i.e., inside ℓ pairs of brackets), a value is any lower level term, or in other words, a term that will have no undelayed region when plugged into a context that supplies ℓ pairs of brackets. Staging annotations use the same nesting rules as LISP's quasiquote and unquote (Dybvig, 1992), but we stress that they preserve scoping: e.g., $\langle \lambda x.\tilde{}(\lambda x.\langle x \rangle)\rangle \equiv \langle \lambda x.\tilde{}(\lambda y.\langle y \rangle)\rangle \not\equiv \langle \lambda y.\tilde{}(\lambda x.\langle y \rangle)\rangle$.

Definition 1 (Erasure)**.** Define erasure $\|e\|$ by the following equations:

$$\|x\| \stackrel{\text{def}}{\equiv} x \qquad \|c\| \stackrel{\text{def}}{\equiv} c \qquad \|\lambda x.e\| \stackrel{\text{def}}{\equiv} \lambda x.\|e\| \qquad \|\tilde{} e\| \stackrel{\text{def}}{\equiv} \|e\|$$
$$\|e_1\ e_2\| \stackrel{\text{def}}{\equiv} \|e_1\|\ \|e_2\| \qquad \|\langle e \rangle\| \stackrel{\text{def}}{\equiv} \|e\| \qquad \|!\, e\| \stackrel{\text{def}}{\equiv} \|e\|$$

We say that a term e is *unstaged* iff $e \equiv \|e\|$, and *staged* otherwise. For example, the `power` function given in the introduction is unstaged, as it is the erasure of `genpow`, which is equal to the erasure of `stpow` modulo η reduction.

The small-step operational semantics is given in Figure 2, where square brackets denote guards on grammatical production rules; e.g.,

$$ECtx_{\mathbf{n}}^{\ell,m} ::= \bullet[m = \ell] \mid \ldots$$

means $\bullet \in ECtx_{\mathbf{n}}^{\ell,m}$ if $m = \ell$. In this semantics, a small-step judgment $e \rightsquigarrow_{\ell} t$ is marked with a level ℓ, which intuitively denotes the number of brackets this step is

Evaluation Contexts (Productions marked $[\phi]$ apply if ϕ holds.)

(CBN) $\mathscr{E}^{\ell,m} \in ECtx_{\mathbf{n}}^{\ell,m} ::= \bullet[m=\ell] \mid \lambda x.\mathscr{E}^{\ell,m}[\ell>0] \mid \langle \mathscr{E}^{\ell+1,m} \rangle \mid {}^{\sim}\mathscr{E}^{\ell-1,m}[\ell>0]$
$\quad \mid\ !\mathscr{E}^{\ell,m} \mid \mathscr{E}^{\ell,m}\ e^{\ell} \mid v^{\ell}\ \mathscr{E}^{\ell,m}[\ell>0] \mid c\ \mathscr{E}^{\ell,m}[\ell=0]$

(CBV) $\mathscr{E}^{\ell,m} \in ECtx_{\mathbf{v}}^{\ell,m} ::= \bullet[m=\ell] \mid \lambda x.\mathscr{E}^{\ell,m}[\ell>0] \mid \langle \mathscr{E}^{\ell+1,m} \rangle \mid {}^{\sim}\mathscr{E}^{\ell-1,m}[\ell>0]$
$\quad \mid\ !\mathscr{E}^{\ell,m} \mid \mathscr{E}^{\ell,m}\ e^{\ell} \mid v^{\ell}\ \mathscr{E}^{\ell,m}$

Substitutable Arguments $a,b \in Arg ::= v^0$ (CBV) $\quad a,b \in Arg ::= e^0$ (CBN)

Small-steps $e^{\ell} \underset{\ell}{\rightsquigarrow} t^{\ell}$ where:

$$\text{SS-}\beta\ \frac{}{(\lambda x.e^0)\ t^0 \underset{0}{\rightsquigarrow} [t^0/x]e^0}\ (\text{CBN}) \qquad \text{SS-}\beta_{\mathbf{v}}\ \frac{}{(\lambda x.e^0)\ v^0 \underset{0}{\rightsquigarrow} [v^0/x]e^0}\ (\text{CBV}) \qquad \text{SS-}\delta\ \frac{(c,d) \in \operatorname{dom}\delta}{c\ d \underset{0}{\rightsquigarrow} \delta(c,d)}$$

$$\text{SS-}E\ \frac{}{{}^{\sim}\langle e^0 \rangle \underset{1}{\rightsquigarrow} e^0} \qquad \text{SS-}R\ \frac{}{!\langle e^0 \rangle \underset{0}{\rightsquigarrow} e^0} \qquad \text{SS-CTX}\ \frac{e^m \underset{m}{\rightsquigarrow} t^m}{\mathscr{E}^{\ell,m}[e^m] \underset{\ell}{\rightsquigarrow} \mathscr{E}^{\ell,m}[t^m]}$$

Fig. 2. Operational semantics of λ^U, parameterized in an interpretation (partial) map $\delta : Const \times Const \rightharpoonup \{v \in V^0 : v \equiv \|v\| \wedge \mathrm{FV}(v) = \varnothing\}$.

happening in. A term takes a small-step at level ℓ iff it decomposes as $\mathscr{E}^{\ell,m}[r]$, where $\mathscr{E}^{\ell,m}$ is an ℓ,m-evaluation context and r is a level-m redex. If the redex r contracts to s, then $\mathscr{E}^{\ell,m}[r] \underset{\ell}{\rightsquigarrow} \mathscr{E}^{\ell,m}[s]$. The SS-CTX rule explicitly requires $s \in E^m$, but this is a purely informative constraint that is always met when the other constraints are satisfied. In general, $e \underset{\ell}{\rightsquigarrow} t$ implies $\ell = \mathrm{lv}\, e \geqslant \mathrm{lv}\, t$ (whose proof we omit).

Redex contractions are: β reduction at level 0, δ reduction at level 0, run-bracket elimination (SS-R) at level 0, and escape-bracket elimination at level 1 (SS-E). All rules are common to both evaluation strategies, except that CBN's β rule is SS-β whereas CBV's is SS-$\beta_{\mathbf{v}}$. The δ reductions are given by a partial map δ from pairs of constants to be closed, unstaged level-0 values, which is undefined for ill-formed pairs like $(\texttt{not}, 5)$. We assume constant applications do not return staged terms.

An ℓ,m-evaluation context $\mathscr{E}^{\ell,m}$ yields a level-ℓ term when plugged with a level-m term. The hole of an evaluation context points to the location of the unique redex that must be contracted next. At level > 0, both evaluation strategies simply walk over the syntax tree of the term to look for escapes, including ones that occur inside the arguments of applications. At level 0, the definition is mostly standard. CBV evaluation contexts can place the hole inside the argument of a level-0 application, whereas CBN evaluation contexts can do so only if the operator is a constant. This difference accounts for the fact that CBV application is always strict at level 0, while CBN application is lazy if the operator is a λ but strict if the operator is a constant.

We use the metavariables $a,b \in Arg$, ranging over the set of substitutable arguments (i.e., e^0 for CBN and v^0 for CBV), to treat both strategies uniformly. For example, the rules SS-β and SS-$\beta_{\mathbf{v}}$ can be unified as

$$\frac{}{(\lambda x.e^0)\ a \underset{0}{\rightsquigarrow} [a/x]e^0}.$$

This semantics is deterministic, and for any ℓ, level-ℓ values cannot take a small-step at level ℓ.

Notation. We write $\lambda^U_{\mathbf{n}} \vdash e \rightsquigarrow_\ell t$ for a CBN small-step judgment and $\lambda^U_{\mathbf{v}} \vdash e \rightsquigarrow_\ell t$ for CBV. We use similar notation for ($\Downarrow$), ($\Uparrow$), and ($\approx$) defined below. We may omit the $\lambda^U_{\mathbf{n}} \vdash$ or $\lambda^U_{\mathbf{v}} \vdash$ if the evaluation strategy either does not matter or is clear from context. For any relation R, let R^+ be its transitive closure and R^* its reflexive–transitive closure.

Definition 2 (Termination and divergence)**.** An $e \in E^\ell$ *terminates* to $v \in V^\ell$ at level ℓ iff $e \rightsquigarrow_\ell^* v$, written $e \Downarrow^\ell v$. We write $e \Downarrow^\ell$ to mean $\exists v.\ e \Downarrow^\ell v$. If no such v exists, then e *diverges* ($e \Uparrow^\ell$). Note that divergence includes stuckness.

Example 3. $p \equiv (\lambda y.\langle 40 + y\rangle)\ (1 + 1)$ is a program. Its value is determined by ($\rightsquigarrow_0$), which works much like in the plain λ calculus. In CBN, $\lambda^U_{\mathbf{n}} \vdash p \rightsquigarrow_0 \langle 40 + (1 + 1)\rangle$. The redex $(1 + 1)$ is not selected for contraction because $(\lambda y.\langle 40 + y\rangle)\ \bullet \notin ECtx^{0,0}_{\mathbf{n}}$. By contrast, in CBV, we have $(\lambda y.\langle 40 + y\rangle)\ \bullet \in ECtx^{0,0}_{\mathbf{v}}$, so $(1 + 1)$ is selected for contraction: $\lambda^U_{\mathbf{v}} \vdash p \rightsquigarrow_0 (\lambda y.\langle 40+y\rangle)\ 2 \rightsquigarrow_0 \langle 40+2\rangle$. Both $\langle 40+(1+1)\rangle$ and $\langle 40+2\rangle$ are level-0 values, and no further small-steps are possible in either evaluation strategy. Thus, $\lambda^U_{\mathbf{n}} \vdash p \Downarrow^0 \langle 40 + (1 + 1)\rangle$ and $\lambda^U_{\mathbf{v}} \vdash p \Downarrow^0 \langle 40 + 2\rangle$.

Example 4. Let $p \equiv \langle \lambda z.z\ (\tilde{}[(\lambda x.x)\ \langle z\rangle])\rangle$, where we used square brackets [ ] *in lieu* of parentheses to improve readability. Let e be the subterm inside the square brackets. In both CBN and CBV, p decomposes as $\mathscr{E}[e]$, where $\mathscr{E} \equiv \langle \lambda z.z\ (\tilde{}\bullet)\rangle \in ECtx^{0,0}$, and e is a level-0 redex. Note the hole of $\mathscr{E}$ is under a binder and the redex e is open, though p is closed. The hole is also in argument position in the application $z\ (\tilde{}\bullet)$ even for CBN. This application is delayed by brackets, so the CBN/CBV distinction is irrelevant in the code generation phase, i.e., until the delay is canceled by !. Hence, $p \rightsquigarrow_0 \langle \lambda z.z\ (\tilde{}\langle z\rangle)\rangle \rightsquigarrow_0 \langle \lambda z.z\ z\rangle$.

Example 5. As an example of evaluation with nested staging constructs, consider $\langle\langle !\,\tilde{}\tilde{}e\rangle\rangle$, where e is some level-0 term that satisfies $e \rightsquigarrow_0 \langle\langle x\rangle\rangle$. Evaluation strategy does not make a difference in this example. We have $\tilde{}\tilde{}e \rightsquigarrow_2 \tilde{}\tilde{}\langle\langle x\rangle\rangle$ because $\tilde{}\tilde{}\bullet \in ECtx^{2,0}$. Moreover, $\tilde{}\langle\langle x\rangle\rangle \rightsquigarrow_1 \langle x\rangle$, so $\tilde{}\tilde{}\langle\langle x\rangle\rangle \rightsquigarrow_2 \tilde{}\langle x\rangle$. However, $\tilde{}\langle x\rangle$ is not a level-2 redex (although it is a level-1 E-redex). The program context $\langle\langle !\,\bullet\rangle\rangle$ is a $0,2$-evaluation context and not a $0,1$-evaluation context, so this $\tilde{}\langle x\rangle$ is not reduced. Thus, $\langle\langle !\,\tilde{}\tilde{}e\rangle\rangle \rightsquigarrow_0^+ \langle\langle !\,\tilde{}\langle x\rangle\rangle\rangle \in V^0$, noting that the contents of the outermost $\langle\bullet\rangle$, namely $\langle !\,\tilde{}\langle x\rangle\rangle$, is a level-0 term. Intuitively, the remaining $\tilde{}$ (as well as !) is delayed by the outermost brackets in $\langle\langle !\,\tilde{}\langle x\rangle\rangle\rangle$.

As usual, this "untyped" formalism can be seen as dynamically typed. In this view, $\tilde{}$ and ! take code-type arguments, where code is a distinct type from functions and base types. Thus, $\langle \lambda x.x\rangle$ 1, $\langle\tilde{}0\rangle$, and !5 are all stuck. Stuckness on variables like x 5 does not arise in programs for conventional languages because programs are closed, but in λ^U, evaluation contexts can pick redexes under binders, so this type of stuckness does become a concern. We will revisit this issue in Section 3. The contraction of open-term level-0 redexes is central to the expressive power of λ^U. It is with this feature that we can evaluate terms like `genpow 3 <x>`, optimizing away the body of the power function.

Remark 1. Binary operations on constants are modeled by including their partially applied variants. To model, say, addition, we take $Const \supseteq \mathbb{Z} \cup \{+\} \cup \{+_k : k \in \mathbb{Z}\}$ and set

$$\delta(+, k) = +_k$$
$$\delta(+_k, k') = (\text{the sum of } k \text{ and } k').$$

For example, using prefix notation, $(+\ 3\ 5) \underset{0}{\rightsquigarrow} (+_3\ 5) \underset{0}{\rightsquigarrow} 8$. Conditionals are modeled by taking $Const \supseteq \{(), \texttt{true}, \texttt{false}, \texttt{if}\}$ and setting

$$\delta(\texttt{if}, \texttt{true}) = \lambda a.\lambda b.a\ ()$$
$$\delta(\texttt{if}, \texttt{false}) = \lambda a.\lambda b.b\ ().$$

Then, e.g., $\texttt{if true}\ (\lambda\_.1)\ (\lambda\_.0) \underset{0}{\rightsquigarrow} (\lambda a.\lambda b.a\ ())\ (\lambda\_.1)\ (\lambda\_.0) \underset{0}{\rightsquigarrow}^* 1$. Pattern-matches on first-order data constructors like lists of integers can be modeled in a similar manner. In the rest of the paper, we will use infix notation and write conditionals as $\texttt{if}\ e_1\texttt{, then}\ e_2\ \texttt{else}\ e_3$ rather than $\texttt{if}\ e_1\ (\lambda\_.e_2)\ (\lambda\_.e_3)$.

The operational semantics induces the usual notion of observational equivalence, which relates terms that are interchangeable under all program contexts. In other words, two expressions are observationally equivalent iff we can silently replace one by the other in any given program without affecting its input/output behavior. This is the sense in which we would like to prove that a staged program like `stpow` is equivalent to its erasure `power`.

Definition 6 (Observational equivalence)**.** Define $e \approx t$ iff for every C such that $C[e], C[t] \in Prog$ we have $C[e] \Downarrow^0 \iff C[t] \Downarrow^0$ and, whenever one of them terminates to a constant, the other also terminates to the same constant.

Remark. This definition of observational equivalence differs from that of the original formulation by Taha (1999) in two respects:

- Taha's definition was stratified, with $(\approx_\ell) \subseteq E^\ell \times E^\ell$ defined for each level ℓ.
- Taha's definition allowed open-term observation, i.e., each $(\approx_\ell)$ was defined just like in Definition 6 but used $C[e], C[t] \in E^0$ in place of $C[e], C[t] \in Prog$.

Because $E^\ell \subseteq E^m$ whenever $\ell \leqslant m$, the $(\approx_\ell)$ at higher levels subsume those at lower levels. The stratification is therefore not terribly useful, and we have dropped it to simplify the notation. Open-term observation is dropped because implementations like MetaOCaml typically reject source files with unbound variables, and closed-term observation more accurately models that design. These changes do not constitute a shift in semantics, however. The old and new definitions give the same equivalence, in the sense that $(\approx) = \bigcup_\ell (\approx_\ell)$. See Appendix A.1 for a proof.

2.2 Equational theory

The equational theory of λ^U is a proof system that, as we will soon show, derives a subset of $(\approx)$. It has four inference rules: compatible extension $(e = t \implies C[e] = C[t])$, reflexivity, symmetry, and transitivity. The CBN axioms are $\lambda^U_{\mathbf{n}} \stackrel{\text{def}}{=}$

$\{\beta, E_U, R_U, \delta\}$, while CBV axioms are $\lambda_{\mathbf{v}}^U \stackrel{\text{def}}{=} \{\beta_{\mathbf{v}}, E_U, R_U, \delta\}$. These axioms are defined below. If $e = t$ can be proved from a set of axioms Φ, then we write $\Phi \vdash e = t$, though we often omit the $\Phi \vdash$ in definitions and assertions that apply uniformly to both CBV and CBN, or if Φ is clear from context. Reduction is a term rewrite induced by the axioms: $\Phi \vdash e \longrightarrow t$ iff $e = t$ is derivable from the axioms by compatible extension alone.

Name	Axiom	Side Condition
β	$(\lambda x.e^0)\ t^0 = [t^0/x]e^0$	
$\beta_{\mathbf{v}}$	$(\lambda x.e^0)\ v^0 = [v^0/x]e^0$	
E_U	$\tilde{}\langle e\rangle = e$	
R_U	$!\langle e^0\rangle = e^0$	
δ	$c\ d = \delta(c,d)$	$(c,d) \in \operatorname{dom}\delta$

Note that $(\longrightarrow)$ is a superset of $\bigcup_\ell (\rightsquigarrow_\ell)$, as the axioms subsume SS-β, SS-$\beta_{\mathbf{v}}$, SS-E, SS-R, and SS-δ, while the inference rule of compatible extension subsumes SS-Ctx. The following example illustrates the difference between reduction and small-steps.

Example 7. Axiom $\beta_{\mathbf{v}}$ gives $\lambda_{\mathbf{v}}^U \vdash (\lambda\_.0)\ 1 = 0$. By compatible extension under $\langle\bullet\rangle$, we have $\langle(\lambda\_.0)\ 1\rangle = \langle 0\rangle$, in fact $\langle(\lambda\_.0)\ 1\rangle \longrightarrow \langle 0\rangle$. Note $\langle(\lambda\_.0)\ 1\rangle \not\rightsquigarrow_0 \langle 0\rangle$ because brackets delay the application; however, reduction allows all left-to-right rewrites by the axioms, so $\langle(\lambda\_.0)\ 1\rangle \longrightarrow \langle 0\rangle$ nonetheless. Intuitively, $\langle(\lambda\_.0)\ 1\rangle \not\rightsquigarrow_0 \langle 0\rangle$ because an evaluator never performs this rewrite, but $\langle(\lambda\_.0)\ 1\rangle \longrightarrow \langle 0\rangle$ because this rewrite is semantics-preserving and a static analyzer or optimizer is allowed to perform it.

Just like the plain λ calculus, λ^U satisfies the Church–Rosser property, so every term has at most one normal form (irreducible reduct). Hence, terms are not provably equal when they have distinct normal forms. Church–Rosser also ensures that reduction and provable equality are more or less interchangeable, and when we investigate the properties of provable equality, we usually do not lose generality by restricting our attention to the simpler notion of reduction.

Theorem 8 (Church–Rosser property)**.** $e = e' \iff \exists t.\ e \longrightarrow^* t \longleftarrow^* e'$.

Next, we establish that provable equality implies observational equivalence.

Theorem 9 (Soundness)**.** $(=) \subset (\approx)$.

The containment $(=) \subset (\approx)$ is proper because $(\approx)$ is not computationally enumerable (since λ^U is Turing-complete) whereas $(=)$ clearly is. There are several useful equivalences in $(\approx) \setminus (=)$, which we will prove by applicative bisimulation. Provable equality is nonetheless strong enough to discover the value of any term that has one, so the assertion "e terminates (at level ℓ)" is interchangeable with "e reduces to a (level-ℓ) value", which in turn is equivalent to "e is provably equal to a (level-ℓ) value".

Theorem 10. If $e \in E^\ell, v \in V^\ell$, then $e \Downarrow^\ell v \implies (e \longrightarrow^* v \wedge e = v)$ and $e = v \implies (\exists u \in V^\ell.u = v \wedge e \longrightarrow^* u \wedge e \Downarrow^\ell u)$.

Theorem 10 is equivalent to the property known as "Plotkin-style correspondence" in the literature, which was shown for the plain λ calculus by Plotkin (1975). It can also be considered a form of the "standardization lemma", although that term usually refers to an equivalence between unrestricted reductions and leftmost, outermost reductions rather than between reductions and evaluations. The proofs of Theorems 8 to 10 can be done with standard, off-the-shelf proof techniques and are therefore omitted. The thesis (Inoue, 2012) contains a proof using Takahashi's technique (1995), which is basically the well-known Tait–Martin-Löf confluence proof using parallel reduction, but extended to also cover standardization.

2.3 Generalized axioms are unsound

This paper's equational theory is not identical to that of Taha (1999), but generalizes rule E_U from $\tilde{}\langle e^0\rangle = e^0$ to $\tilde{}\langle e\rangle = e$. In this subsection, we discuss the utility of this generalization and explain why other axioms cannot be generalized in the same manner.

The main use of the new, generalized E_U is to show that substitution preserves ($\approx$). Thus, an equivalence proved on open terms holds for any closed instance. This fact plays an important role in the completeness proof of applicative bisimulation (see Appendix A.2). It is also somewhat surprising, considering that the converse fails in CBV (see Section 3).

Proposition 11. If $e \approx t$, then $[a/x]e \approx [a/x]t$.

Proof. Take $\ell = \max(\mathrm{lv}\, e, \mathrm{lv}\, t)$. Then, from $e \approx t$, we get

$$(\lambda x.\langle\langle \ldots \langle e\rangle \ldots\rangle\rangle)\ a \approx (\lambda x.\langle\langle \ldots \langle t\rangle \ldots\rangle\rangle)\ a$$

where e and t are each enclosed in ℓ pairs of brackets. Both sides are level 0, so we can apply the β or β_{v} rule, depending on the evaluation strategy, and

$$\langle\langle \ldots \langle [a/x]e\rangle \ldots\rangle\rangle \approx \langle\langle \ldots \langle [a/x]t\rangle \ldots\rangle\rangle.$$

Escaping both sides ℓ times gives

$$\tilde{}\tilde{}\cdots\tilde{}\langle\langle \ldots \langle [a/x]e\rangle \ldots\rangle\rangle \approx \tilde{}\tilde{}\cdots\tilde{}\langle\langle \ldots \langle [a/x]t\rangle \ldots\rangle\rangle.$$

Then, applying the E_U rule ℓ times gives $[a/x]e \approx [a/x]t$. The old E_U rule $\tilde{}\langle e^0\rangle = e^0$ would apply only once here because the level of the $\langle\langle \ldots \langle [a/x]e\rangle \ldots\rangle\rangle$ part increases, so the generalization is strictly necessary. □

At this point, it is natural to wonder why the other rules, β/β_{v} and R_U are not generalized to arbitrary levels, and why E_U is special. The reason is that generalizations of β/β_{v} and R_U involve demotion—moving a term from one level to another. MSP type system researchers have long observed that unrestricted demotion is a type-unsafe operation (Taha & Nielsen, 2003; Westbrook *et al.*, 2010). We show here that it is also unsound as an equational rule.

Table 1 shows generalized rules along with counterexamples that show their unsoundness. The left column names the rule that was generalized, the middle

Table 1. *Generalized equational axioms are unsound.* Ω *is some divergent level-0 term*

Rule name	Generalization	Counterexample	
R_U	$!\langle e\rangle = e$	$\langle !\langle \tilde{}\Omega\rangle\rangle \not\approx \langle \tilde{}\Omega\rangle$	(5)
β	$(\lambda x.e^0)\ t = [t/x]e^0$	$\langle (\lambda x.\langle x\rangle)\ (\lambda y.\tilde{}\Omega)\rangle \not\approx \langle\langle \lambda y.\tilde{}\Omega\rangle\rangle$	(6)
$\beta_{\mathbf{v}}$	$(\lambda x.e^0)\ v^{\ell} = [v^{\ell}/x]e^0$	same as Equation (6)	(7)
$\beta_{\mathbf{v}}$	$(\lambda x.e^0)\ (\lambda y.t) = [(\lambda y.t)/x]e^0$	same as Equation (6)	(8)
$\beta_{\mathbf{v}}$	$(\lambda x.e^0)\ \langle t\rangle = [\langle t\rangle/x]e^0$	$\langle (\lambda x.\langle\langle x\rangle\rangle)\ \langle \tilde{}\tilde{}\Omega\rangle\rangle \not\approx \langle\langle \tilde{}\tilde{}\Omega\rangle\rangle\rangle$	(9)
β	$(\lambda x.e)\ t^0 = [t^0/x]e$	$\langle (\lambda x.\tilde{}x)\ \langle e^0\rangle\rangle \not\approx \langle \tilde{}\langle e^0\rangle\rangle$	(10)
$\beta_{\mathbf{v}}$	$(\lambda x.e)\ v^0 = [v^0/x]e$	same as Equation (10)	(11)

column shows the generalization, and the right column refutes it. Dropping level constraints from R_U gives Equation (5). In CBN β, relaxing the argument's level gives Equation (6). In CBV $\beta_{\mathbf{v}}$, simply removing the argument's level constraint produces $(\lambda x.e^0)\ v^{\ell} = [v^{\ell}/x]e^0$, which is absurd—it subsumes CBN reduction, as $V^1 = E^0$. More sensible attempts are Equations (8) and (9), which keep the constraints on head term constructors. Generalizing the function in β and $\beta_{\mathbf{v}}$ gives Equations (10) and (11), respectively.

Equations (5)–(9) fail because they involve demotion, which moves a term from one level to another. For example, the generalized Equation (5) puts e inside more brackets on the left-hand side than on the right-hand side. The counterexample exploits this mismatch by choosing an e that contains a divergent term enclosed in just enough escapes so that the divergence is forced on one side but not the other. More concretely, on the left-hand side $!\langle \tilde{}\Omega\rangle \in E^0$ so $\langle !\langle \tilde{}\Omega\rangle\rangle \in V^0$. However, on the right-hand side, the Ω is enclosed in fewer brackets and $\langle \tilde{}\Omega\rangle \notin V^0$; in fact $\langle \tilde{}\bullet\rangle \in ECtx^{0,0}$ so assuming $\Omega \underset{0}{\rightsquigarrow} \Omega_1 \underset{0}{\rightsquigarrow} \Omega_2 \underset{0}{\rightsquigarrow} \cdots$ ad infinitum, we have $\langle \tilde{}\Omega\rangle \underset{0}{\rightsquigarrow} \langle \tilde{}\Omega_1\rangle \underset{0}{\rightsquigarrow} \langle \tilde{}\Omega_2\rangle \underset{0}{\rightsquigarrow} \cdots$ as well. We can formalize this insight as follows.

Definition 12 (Level function)**.** Define $\Delta : Ctx \to \mathbb{Z}$ as follows:

$$\Delta \bullet \stackrel{\text{def}}{=} 0 \qquad \Delta(C\ e) \stackrel{\text{def}}{=} \Delta C \qquad \Delta\langle C\rangle \stackrel{\text{def}}{=} \Delta C - 1 \qquad \Delta(!\,C) \stackrel{\text{def}}{=} \Delta C$$
$$\Delta(\lambda x.C) \stackrel{\text{def}}{=} \Delta C \qquad \Delta(e\ C) \stackrel{\text{def}}{=} \Delta C \qquad \Delta(\tilde{}C) \stackrel{\text{def}}{=} \Delta C + 1$$

Proposition 13. There exists a function $L : Ctx \to \mathbb{N}$ such that $\forall e, C.\ \mathrm{lv}\, e \geqslant L(C) \Longrightarrow \mathrm{lv}\, C[e] = \mathrm{lv}\, e + \Delta C$.

Proof. Induction on C. □

Intuitively, ΔC is the limiting value of $\mathrm{lv}\, C[e] - \mathrm{lv}\, e$ as $\mathrm{lv}\, e \to \infty$. This difference converges to a constant independent of e because when e is sufficiently high-level, the deepest nesting of escapes in $C[e]$ occurs within e. Then, $\mathrm{lv}\, C[e] - \mathrm{lv}\, e$ depends only on the number of brackets and escapes surrounding the hole of C. The function L in Proposition 13 gives a lower bound on $\mathrm{lv}\, e$ needed to reach this limiting behavior.

Theorem 14. If $\Delta C \neq \Delta C'$, then $\exists e.\ C[e] \not\approx C'[e]$. That is, a rewrite rule from which we can derive $\forall e.\ C[e] \longrightarrow C'[e]$ for such C and C' is always unsound.

The proof of this theorem relies on the fact that if e has enough escapes, the escapes dominate all the staging annotations in C and the term they enclose is given top priority during program execution. In more technical terms, $\mathrm{lv}\, C[e]$ grows unboundedly with $\mathrm{lv}\, e$ because of Proposition 13, and beyond a certain threshold, $C \in ECtx^{\ell+\Delta C,\ell}$. Hence if, say, $\Delta C > \Delta C'$, then e is evaluated first under C' but not under C. Notice that this proof fails, as expected, if the e in $C[e] \longrightarrow C'[e]$ is restricted to e^0.

Lemma 15 (Context domination). $\mathrm{size}(C) < \ell \Longrightarrow \exists m.\ C \in ECtx^{\ell,m}$.

Proof. Induction on C. □

Lemma 16. $\Delta \mathscr{E}^{\ell,m} = \ell - m$.

Proof. Induction on $\mathscr{E}^{\ell,m}$. □

Lemma 17. If $t^m \Uparrow^m$, then $\mathscr{E}^{\ell,m}[t^m] \Uparrow^\ell$.

Proof. Easily seen from the fact that $\mathscr{E}^{\ell,m}[t^m]$ takes a small step iff t^m does. □

Proof of Theorem 14. Take $\ell \stackrel{\text{def}}{=} \max(L(C), L(C'), \mathrm{size}(C)+1, \mathrm{size}(C')+1)$, where L is a function that witnesses Proposition 13, and let $e \equiv \underbrace{\tilde{}\tilde{} \cdots \tilde{}}_{\ell \text{ times}}\Omega$, where $\Omega \in E^0$ and $\Omega \Uparrow^0$. Then, $\mathrm{lv}\, e = \ell$, $e \Uparrow^\ell$, $\mathrm{lv}\, C[e] = \ell + \Delta C$, and $\mathrm{lv}\, C'[e] = \ell + \Delta C'$. Without loss of generality, $\Delta C > \Delta C'$. By Lemma 15, $C \in ECtx^{\ell+\Delta C,\ell}$, where the second superscript is known by Lemma 16. Then, taking $C_{\langle\cdots\rangle} \stackrel{\text{def}}{\equiv} \langle\langle \cdots \langle \bullet \rangle \cdots \rangle\rangle$ with $\ell + \Delta C$ pairs of brackets, $C_{\langle\cdots\rangle}[C] \in ECtx^{0,\ell}$, so we get $C_{\langle\cdots\rangle}[C[e]] \Uparrow^0$ by Lemma 17. By contrast, $\mathrm{lv}\, C'[e] < \ell + \Delta C$, so $C_{\langle\cdots\rangle}[C'[e]]$ is of the form $\langle t^0 \rangle$, hence $C_{\langle\cdots\rangle}[C'[e]] \Downarrow^0$. □

Theorem 14 provides a quick sanity check for all equational rewrites, which we may call the *level function test*: A rewrite rule must always rewrite between C and C' with $\Delta C = \Delta C'$. In particular, Equations (5) through (9) above fail this test—they rewrite between contexts with different Δ values. Note that a sound rule can rewrite between contexts C and C' such that $\mathrm{lv}\, C[e] - \mathrm{lv}\, e$ and $\mathrm{lv}\, C'[e] - \mathrm{lv}\, e$ disagree for some e, as long as those e are all low level. For example, E_U states $\tilde{}\langle e \rangle = e$, but if $e \in E^0$, then $\mathrm{lv}\, \tilde{}\langle e \rangle - \mathrm{lv}\, e = 1 \neq \mathrm{lv}\, e - \mathrm{lv}\, e$. However, the differences of exact levels agree whenever $\mathrm{lv}\, e \geqslant 1$, which is why Theorem 14 does not apply to E_U. Restricting the level of expressions that can plug level-mismatching holes may also ensure soundness; non-generalized R_U does this.

The Equations (10) and (11) in Table 1 happen to pass the level function test. These rules have in a sense a dual problem: The substitutions in Equations (10) and (11) inject extra brackets to locations that were previously stuck on a variable, whereas Theorem 14 injects extra escapes.

3 Closing substitutions compromise validity

While λ^U is amenable to equational reasoning using β equality, reminiscent of equational reasoning in the plain λ calculus, there is a striking difference in the way

free variables behave in the two settings. This difference is more pronounced in the CBV setting. Traditionally, CBV calculi admit the equational rule

$$(\beta_x)\quad (\lambda y.e^0)\ x = [x/y]e^0$$

Plotkin's seminal λ_V calculus (1975), for example, does so implicitly by taking variables to be values, defining $x \in V$, where V is the set of values for λ_V. But β_x is *not* admissible in $\lambda_{\mathbf{v}}^U$. For example, the terms $(\lambda\_.0)\ x$ and 0 may seem interchangeable, but in $\lambda_{\mathbf{v}}^U$ they are distinguished by the program context $\mathscr{E} \stackrel{\text{def}}{\equiv} \langle \lambda x.\tilde{}[(\lambda\_.\langle 1\rangle)\ \bullet]\rangle$:

$$\lambda_{\mathbf{v}}^U \vdash \langle \lambda x.\tilde{}[(\lambda\_.\langle 1\rangle)\ ((\lambda\_.0)\ x)]\rangle \Uparrow^0 \text{ but } \lambda_{\mathbf{v}}^U \vdash \langle \lambda x.\tilde{}[(\lambda\_.\langle 1\rangle)\ 0]\rangle \Downarrow^0 \langle \lambda x.1\rangle \tag{12}$$

(We are using [ ] as parentheses to enhance readability.) The term on the left is stuck because $x \notin V^0$ and $x \not\xrightarrow[0]{}$. Intuitively, the value of x is demanded before anything is substituted for it, so an implementation would raise an error saying "unbound variable: x". If we apply a substitution σ that replaces x by a value, then $\sigma((\lambda\_.0)\ x) = \sigma 0$, so the standard technique of reasoning under closing substitutions is unsound. Note the β_x redex itself need not contain staging annotations; thus, adding staging to a language can compromise some existing equivalences, i.e., staging is a non-conservative language extension.

The problem here is that $\lambda_{\mathbf{v}}^U$ can evaluate open terms. The reader may recall that λ_V *reduces* open terms just fine while admitting β_x, but the crucial difference is that λ^U *evaluates* (small steps) open terms under program contexts whereas λ_V never does. Small-steps are the specification for implementations, so if they can rewrite an open subterm of a program, implementations must be able to perform that rewrite as well. By contrast, reduction is just a semantics-preserving rewrite, so implementations may or may not be able to perform it.

Implementations of $\lambda_{\mathbf{v}}^U$ including MetaOCaml have no runtime values or data structures representing the variable x—they implement $x \notin V^0$. They never perform $(\lambda\_.0)\ x \xrightarrow[0]{} 0$, for if they were forced to evaluate $(\lambda\_.0)\ x$, then they would try to evaluate the x as required for CBV and throw an exception. Some program contexts in λ^U do force the evaluation of open terms, e.g., the $\mathscr{E}$ given above. We must then define a small-step semantics with $(\lambda\_.0)\ x \not\xrightarrow[0]{} 0$, or else we would not model actual implementations. Moreover, this behavior is conceptually the more natural choice. Variables are placeholders for as-yet-unavailable values, and it makes no sense for the placeholder itself to be offered up as the value. If a reified variable is needed, that is the role of $\langle x\rangle$, not x. Therefore, we must reject β_x, for it is unsound for $(\approx)$ in a small-step semantics with $x \Uparrow^0$. In other words, lack of β_x is an inevitable consequence of the way natural, practical implementations behave.

Even in λ_V, setting $x \in V$ is technically a mistake because λ_V implementations typically do not have runtime representations for variables either. But in λ_V, whether a given evaluator implements $x \in V$ or $x \notin V$ is unobservable. Small steps on a λ_V program (which is closed by definition) never contract open redexes because evaluation contexts cannot contain binders. Submitting programs to an evaluator will never tell if it implements $x \in V$ or $x \notin V$. Therefore, in λ_V, there is never any harm in pretending $x \in V$. A small-step semantics with $x \in V$ gives the same $(\approx)$ as one with $x \notin V$, and β_x is sound for this $(\approx)$.

Intuitively, the reason we can pretend x is a value in λ_V is that by the time execution reaches a subterm with x free, the x will always have a value. Execution only deals with closed instances of terms in the program, so reasoning also only needs to examine closed instances. By contrast, in $\lambda_{\mathbf{v}}^U$ whether a free variable will have a value during execution depends on the context. To be interchangeable under all contexts, terms must behave identically whether all, some, or none of the free variables have values. Thus, *a priori*, comparing terms in λ^U should require comparing under all substitutions, including partial ones. But comparing terms under all substitutions involves comparing under the empty substitution. If we understand "comparing under substitutions" as establishing ($\approx$) under substitutions, then to show $e \approx t$ we would have to show $\varnothing e \approx \varnothing t$, a catch-22.

In an effort to avoid this circularity, one could consider comparing terms by a more lax criterion under the empty substitution than under other substitutions. For example, one might test ($\approx$) under closing substitutions but equi-termination under the empty substitution. That is, to establish $e \approx t$, we check $e \Downarrow^{\ell} \Longleftrightarrow t \Downarrow^{\ell}$ and $\forall$closing σ. $\sigma e \approx \sigma t$. However, these comparisons fail to distinguish between x and y. Free variables are unlike values because they can be divergent, but they are also unlike closed, divergent terms because they are distinguishable, and any attempts to characterize the equivalence between open terms must respect this distinction. Short of considering all the ways in which free variables can be independently substituted for, including not being substituted, there seems to be no clean way to encode this distinction. The applicative bisimulation to be introduced in Section 5 works along this line, considering all substitutions by default, but it allows in some cases to restrict our attention to those substitutions that substitute away some variables.

Thus, the issue with β_x shown above is just the tip of the iceberg. The general, more important, challenge in λ^U is that reasoning under all closing substitutions is insufficient, i.e., ($\forall$closing σ. $\sigma e \approx \sigma t$) $\not\Longrightarrow$ $e \approx t$. We stress that the real challenge is this more general problem with substitutions, and not the special case of β_x, because unfortunately β_x is not only an illustrative example but also a tempting straw man. Seeing β_x alone, one may think that its unsoundness is some idiosyncrasy that can be fixed by modifying the calculus. For example, type systems can easily recover β_x by banishing all stuck terms including β_x redexes. Alternatively, one could modify the implementation (unnaturally, in our opinion) to treat variables as values and define $x \in V^0$, thereby subsuming β_x in β_v. But this little victory over β_x does not matter much, for the general question of when exactly we can reason under closing substitutions remains. It is unclear if any type systems justify reasoning under closing substitutions in general, or how we might be able to prove that.

Surveying which refinements (including, but not limited to the addition of type systems) for λ^U let us reason under substitutions, and why, is an important topic for future study, but it is beyond the scope of this paper. In this paper, we focus instead on showing that we can achieve a lot without committing to anything more complicated than λ^U. In particular, we will show that the lack of β_x is not a large drawback after all, as a refined form of β_x can be proved thanks to applicative

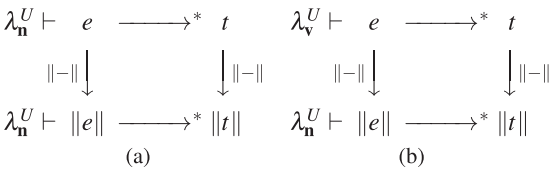

Fig. 3. Visualizations of the Erasure Theorem and the derived correctness lemma. (a) CBN erasure, (b) CBV erasure.

bisimulation (Section 5). The refined rule is

$$(C\beta_x) \quad \lambda x.C[(\lambda y.e^0)\ x] = \lambda x.C[[x/y]e^0]$$

with the side conditions that $C[(\lambda y.e^0)\ x], C[[x/y]e^0] \in E^0$ and that C does not shadow the binding of x. Intuitively, given just the term $(\lambda y.e^0)\ x$, we cannot tell if x is well-leveled, i.e., bound at a lower level than its use, so that a value is substituted for x before evaluation can reach it. The $C\beta_x$ rule remedies this problem by demanding a well-leveled binder. As a special case, β_x is sound for any subterm in the erasure of a closed term—that is, the erasure of any self-contained generator.

4 The erasure theorem

In this section, we present the Erasure Theorem for λ^U and derive simple termination conditions that guarantee $e \approx \|e\|$. The theorem statement differs for CBN and CBV, and the latter has quite a few details to be discussed. We present the simpler CBN first.

4.1 CBN version

The intuition behind the theorem is that all that staging annotations do is describe and enforce an evaluation strategy. They may force CBV, CBN, or some other strategy that the programmer wants, but CBN reduction can simulate any strategy because it allows the redex to be chosen from anywhere.[2] Thus, erasure commutes with CBN reductions (Figure 3(a)). The same holds for provable equalities.

Theorem 18 (CBN Erasure)**.** If $\lambda_{\mathbf{n}}^U \vdash e \longrightarrow^* t$, then $\lambda_{\mathbf{n}}^U \vdash \|e\| \longrightarrow^* \|t\|$. Consequently, if $\lambda_{\mathbf{n}}^U \vdash e = t$, then $\lambda_{\mathbf{n}}^U \vdash \|e\| = \|t\|$.

Proof. The first part is by induction on the derivation of the reduction judgment. The second part follows immediately. □

This theorem gives useful intuitions about what staging annotations can or cannot do in CBN. For example, staging preserves return values up to erasure if those values exist:

[2] This observation does not imply that staging is useless in CBN. It only means that reductions under exotic evaluation strategies are semantics-preserving rewrites under CBN semantics. CBN evaluators may not actually perform such reductions unless forced by staging annotations, which is why staging is interesting in CBN.

Corollary 19. If $u, v \in V^0$ and $(\lambda^U_{\mathbf{n}} \vdash e \longrightarrow^* u \wedge \lambda^U_{\mathbf{n}} \vdash \|e\| \longrightarrow^* v)$, then $v \equiv \|v\|$ and $\lambda^U_{\mathbf{n}} \vdash \|u\| = v$.

Additionally, in CBN, erasure cannot make a term less terminating (equivalently, staging cannot make a term more terminating), unless the annotations affect the term's external interface, that is, unless the staged term's return value carries staging annotations.

Corollary 20. If $\lambda^U_{\mathbf{n}} \vdash e \Downarrow^{\ell} \|v\|$, then $\lambda^U_{\mathbf{n}} \vdash \|e\| \Downarrow^{\ell} \|v\|$.

How does the Erasure Theorem help prove equivalences of the form $e \approx \|e\|$? The theorem gives a simulation of reductions from e by reductions from $\|e\|$. If e reduces to an unstaged term $\|t\|$, then simulating that reduction from $\|e\|$ gets us to $\|\|t\|\|$, which is just $\|t\|$; thus, $e \longrightarrow^* \|t\| \longleftarrow^* \|e\|$ and $e = \|e\|$. Amazingly, this witness $\|t\|$ can be *any* reduct of e, as long as it is unstaged! In fact, by Church–Rosser, any t with $e = \|t\|$ will do. So staging is correct (i.e., semantics-preserving, or $e \approx \|e\|$) if we can find this $\|t\|$. As we will see shortly, this search boils down to a termination check on the generator.

Lemma 21 (CBN correctness)**.** $(\exists t.\ \lambda^U_{\mathbf{n}} \vdash e = \|t\|) \Longrightarrow \lambda^U_{\mathbf{n}} \vdash e = \|e\|$.

4.2 Example: Erasing CBN staged power

Let us show how the Erasure Theorem applies to `stpow`. First, some technicalities: we assume that the *Const* set of λ^U is equipped with integers, arithmetic operators, and booleans, with their usual semantics captured by δ reductions. MetaOCaml's constructs are interpreted in λ^U in the obvious manner, e.g., `let x = e in t` stands for $(\lambda x.t)\ e$ and `let rec f x = e` stands for `let f =` $\Theta(\lambda f.\lambda x.e)$, where Θ is some fixed-point combinator. For conciseness, we treat top-level bindings `genpow` and `stpow` like macros, so $\|$`stpow`$\|$ is the erasure of the recursive function to which `stpow` is bound, with `genpow` inlined, and not the erasure of a variable named `stpow`.

As a caveat, we might wish to prove `stpow` $\approx$ `power`, but unfortunately this goal is unprovable. The whole point of `stpow` is that it processes the first argument without waiting for the second, so it may immediately diverge when partially applied to one argument, whereas `power` does not diverge until it is fully applied. For example, `stpow 0` $\Uparrow^0$ but `power 0` $\Downarrow^0$. We sidestep this issue for now by concentrating on positive arguments, and discuss divergent cases in Section 5.2.

To prove $k > 0 \Longrightarrow$ `stpow` $k =$ `power` k for CBN, we only need to check that the code generator `genpow` k terminates to some `<`$\|e\|$`>`; then the `!` in `stpow` will take out the brackets and we have the witness for applying Lemma 21. To say that something terminates to `<`$\|e\|$`>` roughly means that it is a two-stage program, which is true for almost all uses of MSP that we are aware of. This use of the Erasure Theorem is augmented by the observation $\|$`stpow`$\|$ = `power`—these functions are not quite syntactically equal, the former containing an additional η redex.

Lemma 22. $\lambda^U_{\mathbf{n}} \vdash \|$`stpow`$\|$ = `power`

Proof. Contract the η redex by (CBN) β. □

Proposition 23 (Erasing CBN power). $\forall k \in \mathbb{Z}^+.\ \lambda^U_{\mathbf{n}} \vdash \texttt{stpow}\ k = \texttt{power}\ k$.

Proof. Induction on k gives some e s.t. `genpow k <x>` $= \texttt{<}\|e\|\texttt{>}$, so

$$\begin{aligned}\texttt{stpow}\ k &= \texttt{! <fun x -> ~(genpow } k \texttt{ <x>)>}\\ &= \texttt{! <fun x -> ~<}\|e\|\texttt{>>}\\ &= \texttt{! <fun x -> }\|e\|\texttt{>}\\ &= \texttt{fun x -> }\|e\|\end{aligned}$$

hence, $\texttt{stpow}\ k = \|\texttt{stpow}\|\ k = \texttt{power}\ k$ by Lemmas 21 and 22. □

This proof illustrates our answer to the erasure question in the introduction, for the CBN case. Erasure is semantics-preserving if the generator terminates to $\langle\|e\|\rangle$. What is particularly pleasing about this proof is that it says so little about what e looks like, or what e computes. The only information we track about this generated code is the absence of left-over annotations. Effectively, the concern of reasoning about the annotations is decoupled from the concern of reasoning about what the generated code computes. This simplicity is a major advantage for reasoning about complex generators like LCS (Section 6.1).

4.3 CBV version: Proof by normalization

CBV satisfies a property similar to Theorem 18, but the situation is more subtle. Staging modifies the evaluation strategy in CBV as well, but not all resulting strategies can be simulated in the erasure by CBV reductions, for $\beta_{\mathbf{v}}$ reduces only a subset of β redexes. For example, if $\Omega \in E^0$ is divergent, then $(\lambda\_.0)\ \langle\Omega\rangle \longrightarrow 0$ in CBV, but the erasure $(\lambda\_.0)\ \Omega$ does not CBV-reduce to 0 since Ω is not a value. However, it is the case that $\lambda^U_{\mathbf{n}} \vdash (\lambda\_.0)\ \Omega \longrightarrow 0$ in CBN. In general, erasing CBV reductions gives CBN reductions (Figure 3(b)).

Theorem 24 (CBV Erasure). If $\lambda^U_{\mathbf{v}} \vdash e \longrightarrow^* t$, then $\lambda^U_{\mathbf{n}} \vdash \|e\| \longrightarrow^* \|t\|$. Also, if $\lambda^U_{\mathbf{v}} \vdash e = t$, then $\lambda^U_{\mathbf{n}} \vdash \|e\| = \|t\|$. Note the premise and conclusion use different evaluation strategies.

This theorem has similar ramifications to the CBN Erasure Theorem but with the caveat that they conclude in CBN, despite having premises in CBV. In particular, if e is CBV-equal to an erased term, then $e = \|e\|$ in CBN.

Corollary 25. $(\exists t.\ \lambda^U_{\mathbf{v}} \vdash e = \|t\|) \Longrightarrow \lambda^U_{\mathbf{n}} \vdash e = \|e\|$.

While these results nicely illustrate how staging is a change of evaluation strategy, without further refinement they are not terribly helpful for verification. We still need a way to prove that the program e is equal to $\|e\|$ in CBV. We have two techniques to offer for this purpose: one is to insist that the witness $\|t\|$ terminates to a CBN-normal form, such as a constant, and the other is to exercise some caution in applying $\beta_{\mathbf{v}}$ equalities. The former is conceptually simpler, but the latter

is sometimes more helpful for verifying higher order functions. We discuss proof by normalization in this section, and leave the other idea for Section 4.5.

The idea of proof by normalization is, given e, to show that e and $\|e\|$ CBV-reduce to constants. Then, by chasing the diagram below, we can show $e = \|e\|$ in CBV. Let's say we found some c, d that satisfy the two horizontal CBV equalities. Then, from the top equality, Corollary 25 gives the left vertical one in CBN. As CBN equality subsumes CBV equality, tracing the diagram counterclockwise from the top-right corner gives $\lambda^U_{\mathbf{n}} \vdash c = d$ in CBN. Then, the right vertical equality $c \equiv d$ follows by the Church–Rosser property in CBN. Finally, tracing the diagram clockwise from the top-left corner gives $\lambda^U_{\mathbf{v}} \vdash e = \|e\|$.

$$\begin{array}{lccc} & \lambda^U_{\mathbf{n}} & & \\ & \top & & \\ \lambda^U_{\mathbf{v}} \vdash & e & = & c \\ & \| & & \| \\ \lambda^U_{\mathbf{v}} \vdash & \|e\| & = & d \end{array}$$

Lemma 26 (CBV correctness)**.** If $\lambda^U_{\mathbf{v}} \vdash e = c$ and $\lambda^U_{\mathbf{v}} \vdash \|e\| = d$, then $\lambda^U_{\mathbf{v}} \vdash e = \|e\|$.

Thus, we can prove $e = \|e\|$ in CBV by showing that each side terminates to some constant, *in CBV*. Though we borrowed CBN facts to derive this lemma, the lemma itself leaves no trace of CBN reasoning. Note that we can straightforwardly generalize the lemma by requiring CBV-termination to CBN-normal forms instead of constants, but the generalized statement mixes CBN and CBV reasoning. Because many functions in practice return ground terms when fully applied, we believe the special case above strikes a good balance between generality and simplicity.

4.4 Example: Erasing CBV staged power by normalization

Let us show how the CBV Erasure Theorem applies to `stpow`. The proof is similar to the CBN case, but we need to fully apply both `stpow` and its erasure to confirm that they both reach some constant. The beauty of Lemma 26 is that we do not have to know what those constants are. Just as in CBN, the erasure $\|\texttt{stpow}\|$ is equivalent to `power`, but note this part of the proof uses $C\beta_x$.

Lemma 27. $\lambda^U_{\mathbf{v}} \vdash \|\texttt{stpow}\| \approx \texttt{power}$

Proof. Contract the η expansion by $C\beta_x$. □

Proposition 28 (Erasing CBV power)**.** Suppose $k \in \mathbb{Z}^+$ and $m \in \mathbb{Z}$. Then, we have $\lambda^U_{\mathbf{v}} \vdash \texttt{stpow}\ k\ m \approx \texttt{power}\ k\ m$.

Proof. We stress that this proof works entirely with CBV equalities; we have no need to deal with CBN once Lemma 26 is established. Induction on k gives $\exists e.\ \texttt{genpow}\ k\ \texttt{<x>} = \texttt{<}\|e\|\texttt{>}$ and $[m/\mathrm{x}]\|e\| \Downarrow^0 m'$ for some $m' \in \mathbb{Z}$. We can do so without explicitly figuring out what $\|e\|$ looks like. The case $k = 1$ is easy; for

$k > 1$, the returned code is `<x * ‖e′‖>`, where $[m/\mathtt{x}]\|e'\|$ terminates to an integer by inductive hypothesis, hence so does `<x * ‖e′‖>`. Then,

$$\begin{aligned}\texttt{stpow}\ k\ m &= \texttt{! <fun x -> ~(genpow}\ k\ \texttt{<x>)>}\ m\\ &= \texttt{! <fun x -> }\|e\|\texttt{>}\ m\\ &= [m/x]\|e\| = m' \in \mathit{Const}\end{aligned}$$

Clearly, `power` k m terminates to a constant. By Lemma 27, $\|\texttt{stpow}\|\ k\ m$ also yields a constant, so by Lemma 26, `stpow` $k\ m = \|\texttt{stpow}\|\ k\ m \approx \texttt{power}\ k\ m$. □

This proof illustrates one possible answer to the erasure question in the introduction for CBV: Erasure is semantics-preserving if the staged and unstaged terms terminate to constants in CBV. Showing the latter requires propagating type information and a termination assertion for the generated code. Type information would come for free in a typed system, but it can be easily emulated in an untyped setting. Hence, we see that correctness of staging generally reduces to termination not just in CBN but also in CBV—in fact, the correctness proof is only a slight modification of the termination proof.

4.5 CBV version: Careful erasure

In the last two sections, we have let erasure map CBV equalities to the superset of CBN equalities and performed extra work to show that the particular CBN equalities we derived hold in CBV as well. An alternative approach is to find a subset of CBV equalities that erase to CBV equalities, which is roughly how Yang (2000) handled CBV erasure. This subsection develops this technique in λ^U. The equalities turn out to be more convenient when presented as pairs of equalities than as restrictions of CBV equalities. The result is a trickier, though more versatile, proof method than proof by normalization.

As discussed in Section 4.3, the problem with erasing CBV reductions is that the argument in a $\beta_{\mathbf{v}}$ redex might no longer terminate when erased. To eliminate this case, we might restrict $\beta_{\mathbf{v}}$ to a "careful" variant with a side condition, like

$$(\beta_{\mathbf{v}\Downarrow})\quad (\lambda x.e^0)\ v^0 = [v^0/x]e^0 \text{ provided } \lambda^U_{\mathbf{v}} \vdash \|v^0\| \Downarrow^0$$

If we define a new set of axioms $\lambda^U_{\mathbf{v}\Downarrow} \stackrel{\text{def}}{=} \{\beta_{\mathbf{v}\Downarrow}, E_U, R_U, \delta\}$, then reductions (hence equalities) under this axiom set erase to CBV reductions. However, $\beta_{\mathbf{v}\Downarrow}$ is much too restrictive. It prohibits contracting redexes of the form $(\lambda y.e^0)\ \langle x\rangle$ (note $x \Uparrow^0$), which are ubiquitous—a function as simple as `stpow` already contains one.

Going back to a concrete example is instructive here. As it turned out, $\beta_{\mathbf{v}}$-reducing `genpow` n `<x>` appearing in `stpow` was safe (as evidenced by Proposition 28), despite the `<x>` which has a divergent erasure. Intuitively, the reason is that `stpow` is expected to be used like `stpow` k m, which expands to

$$\texttt{! <fun x -> ~(genpow}\ k\ \texttt{<x>)>}\ m \tag{13}$$

The m is waiting to be substituted for `x`, and indeed it would be substituted right away if it weren't for the staging annotations. Therefore, it is reasonable to exploit

this substitution in checking the side condition for $\beta_{\mathbf{v}\Downarrow}$, because that condition is a check on the behavior of the erasure. Thus, `genpow` k `<x>` should be reduced not by $\beta_{\mathbf{v}\Downarrow}$ but by the refined rule

$$(\beta_{\mathbf{v}\Downarrow}/\sigma) \qquad (\lambda x.e^0)\ v^0 = [v^0/x]e^0 \text{ provided } \lambda^U_{\mathbf{v}} \vdash \sigma\|v^0\| \Downarrow^0$$

with $\sigma = [m/\mathrm{x}]$. This refinement lets us reduce redexes with open-term arguments as long as the σ covers the relevant variables.

An axiom set with $\beta_{\mathbf{v}\Downarrow}/\sigma$ in place of $\beta_{\mathbf{v}}$ can be formulated so that equalities erase as

$$\lambda^U_{\mathbf{v}\Downarrow}/\sigma \vdash e = t \Longrightarrow \lambda^U_{\mathbf{v}} \vdash \sigma\|e\| = \sigma\|t\| \tag{14}$$

The resulting system is strong enough to equate `genpow` k `<x>` to an erased term, using $\sigma = [m/\mathrm{x}]$, but it still falls short of equating `stpow` k to an erased term. The $\beta_{\mathbf{v}\Downarrow}/\sigma$ rule requires the reduction of the staged term to be performed in lockstep with the reduction of the erasure; however, the reduction of expression (13) substitutes m for x at the end whereas the reduction of the erasure `(fun x -> ‖genpow‖ k x)` m substitutes first. In general, the whole point of staging is to reorder the reductions, so we must allow escaping the lockstep at a few strategic places in order to align the rest of the reductions of the staged term and the erasure. To this end, we *define* careful equalities by formula (14) and take $\beta_{\mathbf{v}\Downarrow}/\sigma$ to be a theorem, instead of the other way around.

Definition 29. For any $\sigma : \mathit{Var} \xrightarrow[\text{fin}]{} V^0$, an expression e reduces *carefully modulo* σ to t, written $\lambda^U_{\mathbf{v}\Downarrow}/\sigma \vdash e \longrightarrow t$, iff $\lambda^U_{\mathbf{v}} \vdash e \longrightarrow t$ and $\lambda^U_{\mathbf{v}} \vdash \sigma\|e\| = \sigma\|t\|$. The σ is called the *speculative substitution* accompanying the careful reduction. Careful equalities are defined analogously, using $(=)$ in place of $(\longrightarrow)$.

The rules E_U, R_U, and $\beta_{\mathbf{v}\Downarrow}/\sigma$ are admissible in this system. Compatible extension $(e = t \Longrightarrow C[e] = C[t])$ is not always admissible, for the context C can capture variables in the speculative substitution. This rule must be constrained to avoid variable capture, as shown in CR-Compat below.

Notation. Let $\mathrm{BV}(C)$ stand for the set of variables that C captures, or binds. Given a substitution $\sigma : \mathit{Var} \xrightarrow[\text{fin}]{} E$, let $\mathrm{FV}(\sigma) \stackrel{\text{def}}{=} \mathrm{dom}\,\sigma \cup \left(\bigcup_{x \in \mathrm{dom}\,\sigma} \mathrm{FV}(\sigma x)\right)$.

Remark. $\mathrm{FV}(\sigma)$ is just the "support" of σ in the terminology of nominal logic (Gabbay & Pitts, 2001). Intuitively, it is the set of variables whose names are significant, i.e., renaming them alters the substitution.

Proposition 30. For any $\sigma : \mathit{Var} \xrightarrow[\text{fin}]{} V^0$, the following rules are admissible. The same holds if we replace all occurrences of $(\longrightarrow)$ by $(=)$ and add reflexivity, symmetry, and transitivity.

$$\beta_{\mathbf{v}\Downarrow}/\sigma\ \frac{\lambda^U_{\mathbf{v}} \vdash \sigma\|v^0\| \Downarrow^0}{\lambda^U_{\mathbf{v}\Downarrow}/\sigma \vdash (\lambda x.e^0)\ v^0 \longrightarrow [v^0/e^0]} \qquad E_U\ \frac{}{\lambda^U_{\mathbf{v}\Downarrow}/\sigma \vdash {}^{\sim}\langle e\rangle \longrightarrow e} \qquad R_U\ \frac{}{\lambda^U_{\mathbf{v}\Downarrow}/\sigma \vdash !\,\langle e^0\rangle \longrightarrow e^0}$$

$$\delta\ \frac{(c,d) \in \mathrm{dom}\,\delta}{\lambda^U_{\mathbf{v}\Downarrow}/\sigma \vdash c\ d \longrightarrow \delta(c,d)} \qquad \text{CR-Compat}\ \frac{\lambda^U_{\mathbf{v}\Downarrow}/\sigma \vdash e \longrightarrow t}{\lambda^U_{\mathbf{v}\Downarrow}/\sigma \vdash C[e] \longrightarrow C[t]}\,[\mathrm{BV}(C) \cap \mathrm{FV}(\sigma) = \varnothing]$$

Proof. Reflexivity, symmetry, transitivity, E_U, and R_U are obviously admissible, so we will focus on the other rules. They are special cases of rules for deriving non-careful CBV reductions, so only the $\sigma\|e\| = \sigma\|t\|$ part needs to be shown.

▷ $[\beta_{\mathbf{v}\Downarrow}/\sigma]$ Because the rule's premise gives $\lambda^U_{\mathbf{v}} \vdash \sigma\|v^0\| = u^0$ for some u^0, we have

$$\begin{aligned}\lambda^U_{\mathbf{v}} \vdash \sigma((\lambda x.\|e^0\|)\ \|v^0\|) &= (\lambda x.\sigma\|e^0\|)\ u^0\\ &= [u^0/x](\sigma\|e^0\|)\\ &= [\sigma\|v^0\|/x](\sigma\|e^0\|)\\ &\equiv \sigma\|[v^0/x]e^0\|,\end{aligned}$$

noting that $x \notin \mathrm{FV}(\sigma)$ by Barendregt's variable convention (Barendregt, 1984), so σ commutes with the binder λx.

▷ [CR-Compat] By the side condition, all variables bound in C are fresh for σ, so

$$\sigma(\|C[e]\|) \equiv \sigma(\|C\|[\|e\|]) \equiv (\sigma\|C\|)[\sigma\|e\|]$$

and likewise for $C[t]$. The premise gives $\lambda^U_{\mathbf{v}} \vdash \sigma\|e\| = \sigma\|t\|$, so it follows by compatible extension that $\lambda^U_{\mathbf{v}} \vdash (\sigma\|C\|)[\sigma\|e\|] = (\sigma\|C\|)[\sigma\|t\|]$. □

With these rules, careful reductions can be performed almost like ordinary CBV reductions. The correctness lemma that applies to the result is pleasantly similar to the one for CBN (cf. Lemma 21), with essentially the same proof.

Lemma 31 (Careful CBV correctness)**.** $(\exists t.\ \lambda^U_{\mathbf{v}\Downarrow}/\sigma \vdash e = \|t\|) \Longrightarrow \lambda^U_{\mathbf{v}} \vdash \sigma e = \sigma\|e\|$.

4.6 Example: Erasing CBV staged power by careful erasure

We now demonstrate the correctness of erasing `stpow` in CBV using Lemma 31. The key issue in such proofs is how to introduce the speculative substitution, which is usually where we need to temporarily escape the lockstep of the reductions of the staged term and the erasure. In the case of `! <fun x -> ~(genpow k <x>)> m`, the speculative substitution is $[m/\mathtt{x}]$, speculating the $\beta_{\mathbf{v}}$-substitution of m into the `fun x`.

Alternative Proof of Proposition 28. Let us recall the proposition's statement:

$$\forall k \in \mathbb{Z}^+.\ \forall m \in \mathbb{Z}.\ \lambda^U_{\mathbf{v}} \vdash \mathtt{stpow}\ k\ m \approx \mathtt{power}\ k\ m$$

We have $\|\mathtt{stpow}\| \approx \mathtt{power}$ (Lemma 27), so it suffices to show $\forall k \in \mathbb{Z}^+.\ \forall m \in \mathbb{Z}.\ \lambda^U_{\mathbf{v}} \vdash \mathtt{stpow}\ k\ m \approx \|\mathtt{stpow}\|\ k\ m$. By induction on k, we can show the existence of an e such that

$$\lambda^U_{\mathbf{v}\Downarrow}/[m/\mathtt{x}] \vdash \mathtt{genpow}\ k\texttt{<x>} = \texttt{<}\|e\|\texttt{>} \tag{15}$$

This equality can be proved entirely with the deduction rules in Proposition 30. The required reductions are: δ reductions to simplify `if-then-else` and integer arithmetic, β reductions to simplify the fixed-point operator and pass around the counter k, and β substitutions of `<x>` into the body of `genpow`. Only the last of these needs the speculative substitution.

Now, we need to justify the speculative substitution. Directly applying Lemma 31 to Equation (15) would give an erased equality under the substitution $[m/\mathtt{x}]$, but we want an equality without substitutions. To solve this problem, we will extend Equation (15) to

$$\lambda^{U}_{\mathbf{v}\Downarrow}/\varnothing \vdash \texttt{!<fun x -> ~(genpow } k \texttt{ <x>)> } m = \texttt{(fun x -> } \|e\|\texttt{) } m \tag{16}$$

where $\varnothing$ is the empty substitution. Then, applying Lemma 31 will leave the desired equality with only the empty substitution (or equivalently no substitutions) attached.

To establish Equation (16), we will make a hop outside of the lockstep reduction rules of Proposition 30. That is, we will reason explicitly in $\lambda^{U}_{\mathbf{v}}$, with substitutions applied to terms instead of being kept under $\lambda^{U}_{\mathbf{v}\Downarrow}/$. On the one hand,

$$\begin{aligned}\lambda^{U}_{\mathbf{v}} \vdash \quad & \texttt{!<fun x -> ~(genpow } k \texttt{ <x>)> } m \\ \overset{(15)}{=}\ & \texttt{!<fun x -> } \|e\|\texttt{> } m \\ =\ & \texttt{(fun x -> } \|e\|\texttt{) } m\end{aligned}$$

On the other hand,

$$\begin{aligned}\lambda^{U}_{\mathbf{v}} \vdash \quad & \texttt{(fun x -> (}\|\texttt{genpow}\|\ k\ \texttt{x)) } m \\ =\ & [m/\mathtt{x}](\|\texttt{genpow}\|\ k\ \texttt{x}) \\ \overset{(15)}{=}\ & [m/\mathtt{x}]\|e\| \\ =\ & \texttt{(fun x -> } \|e\|\texttt{) } m\end{aligned}$$

Therefore, Equation (16) holds by the definition of careful equalities, whence we immediately get

$$\lambda^{U}_{\mathbf{v}\Downarrow}/\varnothing \vdash \texttt{stpow } k\ m = \texttt{(fun x -> } \|e\|\texttt{) } m$$

The right-hand side is unstaged, so by Lemma 31,

$$\lambda^{U}_{\mathbf{v}} \vdash \texttt{stpow } k\ m = \|\texttt{stpow}\|\ k\ m$$ □

Overall, the analyses involved in proof by normalization and proof by careful equalities are quite similar. In both approaches, we track the reduction of the generated code while reducing the generator, which requires tracking the substitution that will be applied when the generated code runs. The normalization approach exploits the substitutions when analyzing the termination of code returned by the generator, whereas careful equalities exploits them when analyzing the termination of code that is passed into the generator.

Both approaches have advantages and disadvantages. The normalization approach can be done entirely in the well-behaved reduction system of $\lambda^{U}_{\mathbf{v}}$, and it does not require us to explicitly relate the staged term's execution with that of the erasure at all. These properties make the approach easier to use, but in exchange, we must supply enough context to force the return value to be a constant. The careful reduction approach does not offer a crisp deductive system but provides only a set of admissible rules reminiscent of $\lambda^{U}_{\mathbf{v}}$'s reduction system (Proposition 30). Those rules suffice for a large part of the reasoning, but some details must be filled in by devising *ad-hoc* arguments for justifying the speculative substitution. In return,

careful equalities do not require getting down to ground values, which as we shall see in Section 6.2 is a notable advantage for verifying higher order generators.

5 Extensional reasoning by applicative bisimulation

This section presents applicative bisimulation, a well-established tool for analyzing higher order functional programs (Abramsky, 1990; Gordon, 1999). Bisimulation is sound and complete for $(\approx)$, in particular justifying $C\beta_x$ (Section 3) and extensionality, allowing us to handle the divergence issues we glossed over in Section 4.2.

5.1 Proof by bisimulation

Here, we present the definition and usage of applicative bisimulation in λ^U and leave the proofs of soundness and completeness to Appendix A.2. Due to technical complications, the *indexed* applicative bisimilarity defined in that appendix, which coincides with observational equivalence, is notationally dense and unwieldy. Therefore, in this section, we work with a reasoning principle packaged up more conveniently, which hides the indexing. The packaged principle is also enhanced (Pous & Sangiorgi, 2011) up to observational equivalence, i.e., bisimulations can contain pairs of terms that transition to terms that are in the bisimulation only modulo observational equivalence. We say "applicative bisimulation" to denote this unindexed, enhanced relation.

For a pair of terms to be applicatively bisimilar, they must both terminate or both diverge. If they terminate, their values must be bisimilar again under experiments that examine their behavior. In an experiment, functions are called, code values are run, and constants are left untouched. Effectively, this is a bisimulation under the labeled transition system consisting of evaluation $(\Downarrow)$ and experiments. If $e\ R\ t$ implies either that $e \approx t$ or that e and t are bisimilar, then $R \subseteq (\approx)$.

Definition 32 (Relation under experiment)**.** Given a relation $R \subseteq E \times E$, let $\widetilde{R} \stackrel{\text{def}}{=} R \cup (\approx)$. Define $R^{\ell}_{\dagger} \subseteq V^{\ell} \times V^{\ell}$ by

$$\frac{}{c\ R^0_\dagger\ c} \qquad \frac{\forall a.\ ([a/x]e^0)\ \widetilde{R}\ ([a/x]t^0)}{(\lambda x.e^0)\ R^0_\dagger\ (\lambda x.t^0)} \qquad \frac{e^0\ \widetilde{R}\ t^0}{\langle e^0\rangle\ R^0_\dagger\ \langle t^0\rangle} \qquad \frac{u^{\ell+1}\ \widetilde{R}\ v^{\ell+1}}{u^{\ell+1}\ R^{\ell+1}_\dagger\ v^{\ell+1}}$$

Definition 33. The *substitution closure* of a binary relation $R \subseteq E \times E$, written $R^\bullet$, is defined as $R^\bullet \stackrel{\text{def}}{=} \{(\sigma e, \sigma t) : e\ R\ t \wedge (\sigma : Var \underset{\text{fin}}{\rightharpoonup} Arg)\}$. A binary relation is *substitution-closed* iff it equals its own substitution closure.

Definition 34 (Applicative bisimulation)**.** A substitution-closed binary relation $R \subseteq E \times E$ is an *applicative bisimulation* iff every $(e,t) \in R$ satisfies the following: Letting $\ell = \max(\mathrm{lv}\, e, \mathrm{lv}\, t)$, we have $e \Downarrow^\ell \iff t \Downarrow^\ell$, and if $e \Downarrow^\ell u \wedge t \Downarrow^\ell v$, then $u\ R^\ell_\dagger\ v$.

Theorem 35. For a substitution-closed binary relation $R \subseteq E \times E$, we have $R \subseteq (\approx)$ iff R is contained in an applicative bisimulation.

In particular, $(\approx)$ is an applicative bisimulation—the largest one under set inclusion, called applicative bisimilarity. Thus, the observably equivalent pairs of

terms are precisely the pairs that are applicatively bisimilar. This is our answer to the extensional reasoning question in the introduction: Bisimulation can in principle derive *all* valid equivalences, including all extensional facts. Unlike in single-stage languages (Abramsky, 1990; Howe, 1996; Gordon, 1999), σ ranges over non-closing substitutions, which may not substitute for all variables or may substitute open terms. Closing substitutions are unsafe because λ^U has open-term evaluation. But for CBV, bisimulation gives a condition under which substitution is safe, i.e., when the binder is at level 0 (in the definition of $(\lambda x.e)\ R^0_\dagger\ (\lambda x.t)$). In CBN, this is not an advantage as $\forall a.[a/x]e\ \widetilde{R}\ [a/x]t$ entails $[x/x]e\ \widetilde{R}\ [x/x]t$, but bisimulation still gives a more approachable alternative to $(\approx)$.

The importance of the substitution in the definition of $(\lambda x.e)\ R^0_\dagger\ (\lambda x.t)$ for CBV is best illustrated by the proof of extensionality, from which we get $C\beta_x$ introduced in Section 3.

Proposition 36. If $e, t \in E^0$ and $\forall a.\ (\lambda x.e)\ a \approx (\lambda x.t)\ a$, then $\lambda x.e \approx \lambda x.t$.

Proof. Take $R \stackrel{\text{def}}{=} \{(\lambda x.e, \lambda x.t)\}^\bullet$. To see that R is a bisimulation, fix σ, and note that $\sigma\lambda x.e$, $\sigma\lambda x.t$ terminate to themselves at level 0. By the variable convention (Barendregt, 1984), x is fresh for σ, so $\sigma\lambda x.e \equiv \lambda x.\sigma e$ and $\sigma\lambda x.t \equiv \lambda x.\sigma t$. We must check $[a/x]\sigma e\ \widetilde{R}\ [a/x]\sigma t$. By assumption and by Proposition 11, we get $\sigma[a/x]e \approx \sigma[a/x]t$, and one can show that σ and $[a/x]$ commute modulo $(\approx)$. Hence, by Theorem 35, $\lambda x.e \approx \lambda x.t$. □

Corollary 37 (Soundness of $C\beta_x$)**.** If $C[(\lambda y.e^0)\ x], C[[x/y]e^0] \in E^0$ and C does not bind x, then $\lambda x.C[(\lambda y.e^0)\ x] \approx \lambda x.C[[x/y]e^0]$.

Proof. Apply both sides to an arbitrary a and use Proposition 36 with $\beta/\beta_\mathbf{v}$. □

Our proof of Proposition 36 would have failed in CBV if we had defined $(\lambda x.e)\ R^0_\dagger\ (\lambda x.t) \iff e\ \widetilde{R}\ t$, without the substitution. For when $e \equiv (\lambda\_.0)\ x$ and $t \equiv 0$, the premise $\forall a.[a/x]e \approx [a/x]t$ is satisfied but $e \not\approx t$, so $\lambda x.e$ and $\lambda x.t$ are not bisimilar with this weaker definition. The binding in $\lambda x.e \in E^0$ is guaranteed to be well-leveled, and exploiting it by inserting $[a/x]$ in the comparison is strictly necessary to get a complete (as in "sound and complete") notion of bisimulation.

Function extensionality is a common addition to the equational theory of the plain λ calculus, usually called the ω rule (Plotkin, 1974; Intrigila & Statman, 2009). But unlike ω in the plain λ calculus, λ^U functions must agree on open-term arguments and not just on closed-term arguments. This is no surprise given λ^U functions do receive open arguments during program execution; however, we know of no specific functions that fail to be equivalent because of open arguments. Whether extensionality can be strengthened to require equivalence only under closed arguments is an interesting open question.

Another important fact which can be proved with applicative bisimulation is that two divergent terms are equivalent. An exception has to be made for the case where one term gets stuck on a free variable while the other diverges for a different reason, but a difference of this kind can be detected by comparing the terms under

substitutions. This result will let us show that `stpow` is interchangeable with its erasure not just in terminating cases but also in non-terminating cases.

Notation. Let $e \approx_{\Uparrow} t$ mean $e \approx t \vee (e{\Uparrow}^{\ell} \wedge t{\Uparrow}^{\ell})$, where $\ell = \max(\mathrm{lv}\, e, \mathrm{lv}\, t)$.

Lemma 38. For a fixed e, t, if for every $\sigma : Var \xrightarrow[\text{fin}]{} Arg$ we have $\sigma e \approx_{\Uparrow} \sigma t$, then $e \approx t$.

Proof. Notice that $\{(e,t)\}^{\bullet}$ is an applicative bisimulation. □

Remark. The only difference between Definition 34 and applicative bisimulation in the plain λ calculus is that Definition 34 avoids applying closing substitutions. Given that completeness can be proved for this bisimulation, it seems plausible that the problem with reasoning under substitutions is the only thing that makes conservativity fail. Hence, it seems that, for *closed* unstaged terms, λ^U's ($\approx$) could actually coincide with that of the plain λ calculus. Such a result would make a perfect complement to the Erasure Theorem, for it lets us completely forget about staging when reasoning about erased programs.

We do not have a proof of this conjecture, however. Conservativity results for observational equivalences are often proved by semantic arguments that exploit denotational models (Mitchell, 1993; Riecke & Subrahmanyam, 1994; McCusker, 2003), but giving such a model for hygienic MSP is notoriously difficult (Benaissa *et al.*, 1999). Although Riecke & Subrahmanyam (1994) do also discuss a more syntactic approach, that proof also occasionally uses semantic arguments. Investigating whether such techniques can be made to work for λ^U deserves consideration in a separate paper.

5.2 Example: Tying loose ends on staged power

In Section 4.2, we sidestepped issues arising from the fact that `stpow 0` $\Uparrow^0$ whereas `power 0` $\Downarrow^0$. If we are allowed to modify the code, this problem is usually easy to avoid, for example, by making `power` and `genpow` return dummy return values for non-positive arguments. If not, we can still persevere by finessing the statement of correctness. The problem is partial application, so we can force `stpow` to be fully applied before it executes by stating `power` $\approx \lambda n.\lambda x.$`stpow` $n\ x$.

Proposition 39 (CBN `stpow` is Correct)**.** $\lambda^U_{\mathbf{n}} \vdash$ `power` $\approx \lambda n.\lambda x.$`stpow` $n\ x$.

Proof. We just need to show $\forall e, t \in E^0.$ `power` $e\ t \approx_{\Uparrow}$ `stpow` $e\ t$, because then $\forall e, t \in E^0.\ \forall \sigma : Var \xrightarrow[\text{fin}]{} Arg.\ \sigma($`power` $e\ t) \approx_{\Uparrow} \sigma($`stpow` $e\ t)$, whence `power` $\approx \lambda n.\lambda x.$`stpow` $n\ x$ by Lemma 38 and extensionality. So fix arbitrary, potentially open, $e, t \in E^0$, and split cases on the behavior of e. As evident from the following argument, the possibility that e, t contain free variables is not a problem here.

- ▷ [If $e \Uparrow^0$ or $e \Downarrow^0 u \notin \mathbb{Z}^+$] Both `power` $e\ t$ and `stpow` $e\ t$ diverge.
- ▷ [If $e \Downarrow^0 m \in \mathbb{Z}^+$] Using Proposition 23, `power` $e =$ `power` $m \approx$ `stpow` $m =$ `stpow` e, so `power` $e\ t \approx$ `stpow` $e\ t$. □

Proposition 40 (CBV `stpow` is correct)**.** $\lambda^U_{\mathbf{v}} \vdash$ `power` $\approx \lambda n.\lambda x.$`stpow` $n\ x$.

Proof. By the same argument as in CBN, we just need to show `power` $u\ v \approx_{\Uparrow}$ `stpow` $u\ v$ for arbitrary $u, v \in V^0$.

- ▷ [If $u \notin \mathbb{Z}^+$] Both `power` $u\ v$ and `stpow` $u\ v$ get stuck at `if n = 0`.
- ▷ [If $u \in \mathbb{Z}^+$] If $u \equiv 1$, then `power` $1\ v = v =$ `stpow` $1\ v$. If $u > 1$, we show that the generated code is strict in a subexpression that requires $v \in \mathbb{Z}$. Observe that `genpow` u `<x>` $\Downarrow^0$ `<`e`>`, where e has the form `<x * t>`. For $[v/x]e \Downarrow^0$, it is necessary that $v \in \mathbb{Z}$. It is clear that `power` $u\ v \Downarrow^0$ requires $v \in \mathbb{Z}$. So either $v \notin \mathbb{Z}$ and `power` $u\ v \Uparrow^0$ and `stpow` $u\ v \Uparrow^0$, in which case we are done, or $v \in \mathbb{Z}$ in which case Proposition 28 applies. □

Remark. Real code should not use $\lambda n.\lambda x.$`stpow` $n\ x$, as it regenerates and recompiles code upon every invocation. Application programs should always use `stpow`, and one must check (outside of the scope of verifying the function itself) that `stpow` is always eventually fully applied so that the η expansion is benign.

6 Case studies

In this section, we verify two concrete generators that are more illustrative of the techniques used in realistic applications than `power` to demonstrate that this article's approach can cover more complex generators. Each example illustrates specific complicating factors that can arise in practical generators:

- The LCS (Section 6.1) couples monadic memoization with continuation-passing style and `let`-insertion (Swadi *et al.*, 2006). This technique is essential for generating code of acceptable quality, but it complicates the generated code. Nonetheless, the proof strategy remains roughly the same.
- The staged fold function (Section 6.2) is an example of a higher order generator—one that takes another generator as input. Despite the fact that the normalization approach (Lemma 26) demands ground terms, we demonstrate that it can also handle higher order code. We also show that, in this context, careful equalities can give a more natural characterization of correctness than the normalization approach.

These examples illustrate that our techniques apply to a wide range of generators, and that we need not hold back on sophisticated programming techniques in order to make the program amenable to analysis.

6.1 Longest common subsequence

In this section, we work out the correctness proof of LCS. Although this example is not practically useful but rather chosen for ease of explanation, much like `power`, its structure is representative of programs exploiting the monadic memoization technique useful for staging a wide range of memoized functions (Swadi *et al.*, 2006). The technique has an effect similar to performing common subexpression elimination on the generated code, although it differs in that the common subexpressions are detected and shared as generation progresses. Compilers, by contrast, usually

perform this optimization in a post-generation pass that inspects the syntactic structure of generated code. This difference notwithstanding, the technique has proved indispensable for compiler-like applications of MSP, such as domain-specific language implementation (Brady & Hammond, 2006; Taha, 2008) and circuit generation (Kiselyov & Taha, 2005).

6.1.1 The code

The code for LCS is displayed in Figure 4. We have several versions of the same function, which maps integers i, j, and 0-based arrays P, Q to the length of the LCS of P and Q. For simplicity, we compute the length of LCS instead of the actual sequence, but modifying it to return the sequence is straightforward. The function `naive_lcs` is a naïve exponential-time implementation serving as the specification, while `lcs` is the textbook polynomial-time version with memoization. The function `stlcs` is the staged version of `lcs` that we wish to verify, which specializes `lcs` to the lengths i and j of the input sequences. All recursive calls in `lcs` and `stlcs` go through memoizing combinators `mem` and `memgen`, respectively.

Memo tables are represented by functions mapping a key k and functions f, g to $f\ v$ if a value v is associated with k, or to $g\ ()$ if k is not in the table. The value `empty` is the empty table, `ext` extends a table with a given key-value pair, and `lookup` looks up the table. This interface is chosen to make the correspondence with λ^U clear. In MetaOCaml, `lookup` can return an option type, but since we left out higher order constructors from λ^U we Church-encoded the option type here. *Const* covers first-order types like `int option`, but not higher order types like `(int -> int) option` or `(int code) option`.

We use a state-continuation monad to hide memo-table passing and to put the code into CPS. Computation in this monad takes a state (the memo table) and a continuation, and calls the continuation with an updated state and return value. All of the effectful computation happens inside the memoization combinators: Both `mem` and `memgen` look up the memo table, call the memoized function if no suitable value is cached, and update the memo table.

6.1.2 Purpose of CPS

The purpose of continuation-passing style translation in LCS is to implement the binding-time improvements first discovered by Bondorf (1992). This section briefly reviews the significance of this improvement. See Swadi *et al.* (2006) for a more thorough treatment.

The crux of the monadic memoization technique is the following part of `memgen`:

```
<let z = ~r in
~(k (ext tab (i,j) <z>)                                    (17)
   <z>)>
```

This is executed precisely when the key `(i,j)` is not in the table `tab`. The variable `r` holds a new code value returned from `genlcs` to be associated with the key `(i,j)`,

```
let ext table key v =
  fun key' f g ->
    if key = key' then f v
    else table key' f g
and empty key f g = g ()
and lookup table key f g =
  table key f g

let ret a = fun s k -> k s a
and bind m f = fun s k ->
  m s (fun s r -> f r s k)
let eval_m m =
  m empty (fun s r -> r)

let rec genlcs i j p q =
 if (i < 0 || j < 0)
 then ret <0>
 else
  bind (memgen genlcs (i-1) (j-1)
                p q) (fun n1 ->
  bind (memgen genlcs (i-1) j
                p q) (fun n2 ->
  bind (memgen genlcs i     (j-1)
                p q) (fun n3 ->
    ret <if (~p).(i) = (~q).(j)
         then ~n1 + 1
         else max ~n2 ~n3>)))
and memgen f i j p q  =
 fun tab k ->
   lookup tab (i,j)
     (fun r -> k tab r)
     (fun _ ->
       f i j p q tab (fun tab r ->
         <let z = ~r in
         ~(k (ext tab (i,j) <z>)
             <z>)>)))
let stlcs i j =
  ! <fun p q ->
      ~(eval_m (genlcs i j
                 <p> <q>))>
```

```
let rec naive_lcs i j p q =
  if (i < 0 || j < 0) then 0
  else
    let n1 = naive_lcs (i-1) (j-1)
                       p q in
    let n2 = naive_lcs (i-1) j
                       p q in
    let n3 = naive_lcs i     (j-1)
                       p q in
      if p.(i) = q.(j) then n1 + 1
      else max n2 n3

let rec lcs_rec i j p q =
  if (i < 0 || j < 0) then ret 0
  else
    bind (mem lcs_rec (i-1) (j-1)
              p q) (fun n1 ->
    bind (mem lcs_rec (i-1) j
              p q) (fun n2 ->
    bind (mem lcs_rec i     (j-1)
              p q) (fun n3 ->
      ret (if p.(i) = q.(j)
           then n1 + 1
           else max n2 n3))))

and mem f i j p q =
  fun tab k ->
    lookup tab (i,j)
      (fun r -> k tab r)
      (fun _ ->
         f i j p q tab (fun tab r ->
           let z = r in
           (k (ext tab (i,j) z) z)))

let lcs i j p q =
  eval_m (lcs_rec i j p q)
```

Fig. 4. Staged and unstaged code for longest common subsequence.

and the `k` is the continuation to the call to `memgen genlcs`, which generates further code using the code associated with `(i,j)`.

Instead of directly registering `r` with `tab` as the value corresponding to `(i,j)`, the code above binds `r` to a new level-1 variable `z`, then registers `<z>` instead. Without this trick, the code inserted into the memo table snowballs exponentially. Suppose we modified `memgen` so that the code in listing (17) is replaced by

```
k (ext tab (i,j) r) r
```

Let $\langle e_{ij} \rangle$ be the code generated for a given i, j pair by `genlcs` using this modified `memgen`. Then, e_{ij} would be as follows, containing $e_{(i-1)(j-1)}$, $e_{(i-1)j}$, and $e_{i(j-1)}$ as subterms:

$$\begin{array}{l} \texttt{if p.(}i\texttt{) = q.(}j\texttt{)} \\ \texttt{then } e_{(i-1)(j-1)} \texttt{+ 1} \\ \texttt{else max } e_{(i-1)j} \texttt{ + } e_{i(j-1)} \end{array} \tag{18}$$

But $e_{(i-1)(j-1)}$ in turn contains $e_{(i-2)(j-2)}$, $e_{(i-2)(j-1)}$, and $e_{(i-1)(j-2)}$, and likewise for $e_{(i-1)j}$ and $e_{i(j-1)}$. The code size is hence exponential in i and j.

Code (17) solves this problem by generating a binding `let` z_{ij} `=` e_{ij} for each i, j, passing on $\langle z_{ij} \rangle$ to be used in place of $\langle e_{ij} \rangle$. Overall, the generated code looks like:

$$\begin{array}{l} \ldots \\ \texttt{let } z_{(i-1)(j-1)} \texttt{ = if ... then } z_{(i-2)(j-2)} \texttt{+ 1 else max } z_{(i-2)(j-1)}\ z_{(i-1)(j-2)} \texttt{ in} \\ \texttt{let } z_{(i-1)j} \texttt{ = if ... then } z_{(i-2)(j-1)} \texttt{+ 1 else max } z_{(i-2)j}\ z_{(i-1)(j-1)} \texttt{ in} \\ \texttt{let } z_{i(j-1)} \texttt{ = if ... then } z_{(i-1)(j-2)} \texttt{+ 1 else max } z_{(i-1)(j-1)}\ z_{i(j-2)} \texttt{ in} \\ \texttt{let } z_{ij} \texttt{ = if ... then } z_{(i-1)(j-1)} \texttt{+ 1 else max } z_{(i-1)j}\ z_{i(j-1)} \texttt{ in} \\ z_{ij} \end{array}$$

For each i, j pair, the variable z_{ij} is bound to Equation (18), but with subterms e_{ij} replaced by variable references. Thus, the right-hand side of the `let` has the same size regardless of i, j. Because a `let` is generated only when memo-table lookup fails, the e_{ij} is bound at most once for every i, j pair, ensuring the code size is polynomial.

Thus, monadic memoization is essential for generating code of acceptable quality. But the generated code takes much more work to describe formally, as one must predict and spell out the order of `let` bindings appearing in the generated code. Fortunately, like with the `power` example, erasure makes such details largely irrelevant, for it lets us get away with rather coarse characterizations of the generated code.

6.1.3 *Correctness proof*

We now sketch the main parts of the correctness proof for LCS. We focus on the harder CBV case and leave CBN as an exercise. We adopt the proof-by-normalization approach, although careful reductions could also work. Let us assume *Const* has unit, booleans, integers, tuples of integers, and arrays thereof with 0-based indices. The symbol $\mathfrak{A}$ stands for the set of all arrays (a subset of *Const*), σ ranges over substitutions $Var \xrightarrow[\text{fin}]{} V^0$, and $e \Downarrow^0 \mathbb{Z}$ means $\exists n \in \mathbb{Z}.\ e \Downarrow^0 n$.

Despite all the complications introduced by monadic memoization, our strategy remains the same as for `power`: check termination and apply the Erasure Theorem. Just like with `power`, to show the termination of `stlcs` we track the invariant that the generated code terminates under suitable substitutions. The difference is that the set of variables that the "suitable substitution" has to cover grows as more `let` bindings are generated.

The invariant is captured in two parts, one for the memo table and one for the continuation. For the memo table, every key should be mapped to some $\langle z \rangle$, where

z should have an integer value under the substitution that will be in effect when the generated code is run.

Definition 41. A *good memo table* is a $T \in E^0$ such that for every $i, j \in \mathbb{Z}$ and every $f, g \in V^0$, either $\lambda_{\mathbf{v}}^U \vdash \texttt{lookup}\ T\ (i,j)\ f\ g = f\ \texttt{<}z\texttt{>}$ for some z or $\lambda_{\mathbf{v}}^U \vdash \texttt{lookup}\ T\ (i,j)\ f\ g = g\ \texttt{()}$. The set of all good memo tables is written G. A good memo table T is *covered by* σ iff σ is a substitution such that for all of the z's we have $z \in \operatorname{dom}\sigma$ and $\sigma z \in \mathbb{Z}$. The set of all good memo tables covered by σ is written G_σ.

The continuation should then preserve termination under substitution, i.e., it should map terminating code $\texttt{<}\|e\|\texttt{>}$ to terminating code $\texttt{<}\|t\|\texttt{>}$. As a caveat, the continuation will be invoked under `lets`, like the call to `k` in code listing (17), so the termination of $\|e\|$ must be assessed under substitutions that cover more variables than were visible when the continuation was created. In the following definition, $\sigma' \supseteq \sigma$ means σ' is an extension of σ (i.e., $\operatorname{dom}\sigma' \supseteq \operatorname{dom}\sigma$ and $\sigma'|_{\operatorname{dom}\sigma} = \sigma$).

Definition 42. Let the set K_σ of all *good continuations under* σ consist of all $k \in V^0$ s.t. for any e, $\sigma' \supseteq \sigma$, and $T \in G_{\sigma'}$ with $\sigma'\|e\| \Downarrow^0 \mathbb{Z}$, we have $\exists t.\ k\ T\ \texttt{<}\|e\|\texttt{>} = \texttt{<}\|t\|\texttt{>}$ and $\sigma'\|t\| \Downarrow^0 \mathbb{Z}$.

With these invariants, the following lemma can be proved: Given a memo table and a continuation that respect these invariants, `genlcs` returns terminating code.

Lemma 43. Fix σ and $T \in G_\sigma$ and $i, j \in \mathbb{Z}$ and $\mathtt{p}, \mathtt{q} \in \mathit{Var}$, such that $\sigma\mathtt{p}, \sigma\mathtt{q} \in \mathfrak{A}$ and $i < \operatorname{length}(\sigma\mathtt{p}) \wedge j < \operatorname{length}(\sigma\mathtt{q})$. Then, $\forall k \in K_\sigma.\ \exists e.\ \sigma\|e\| \Downarrow^0 \mathbb{Z}$ and

$$\lambda_{\mathbf{v}}^U \vdash \texttt{genlcs}\ i\ j\ \texttt{<p>}\ \texttt{<q>}\ T\ k = \texttt{<}\|e\|\texttt{>}$$

Proof. Lexicographic induction on (i, j). □

Now, we can prove the main theorem, the correctness of LCS.

Theorem 44. $\lambda_{\mathbf{v}}^U \vdash \texttt{naive\_lcs} \approx \lambda x.\lambda y.\lambda p.\lambda q.\texttt{stlcs}\ x\ y\ p\ q$

Proof. By extensionality and Lemma 38, it suffices to prove

$$\texttt{naive\_lcs}\ i\ j\ P\ Q \approx_{\Uparrow} \texttt{stlcs}\ i\ j\ P\ Q$$

for every $i, j, P, Q \in V^0$. Here, we focus on the case where both sides converge, that is,

$$i, j \in \mathbb{Z} \qquad \text{and} \qquad P, Q \in \mathfrak{A} \qquad \text{and} \qquad i < \operatorname{length}(P) \wedge j < \operatorname{length}(Q)$$

leaving the less interesting cases to the thesis (Inoue, 2012). Under these assumptions, `stlcs` $i\ j\ P\ Q$ reduces to

$$\texttt{! <fun p q ->\ (genlcs}\ i\ j\ \texttt{<p> <q> empty (fun s r -> r))>}\ P\ Q \tag{19}$$

Let $\sigma = [P, Q/\mathtt{p}, \mathtt{q}]$. Then, it is easily seen that $\texttt{empty} \in G_\sigma$ and $\texttt{(fun s r -> r)} \in K_\sigma$, so by Lemma 43, there exists a term $\|e\|$ with $\sigma \Downarrow^0 \mathbb{Z}$ such that

$$\begin{aligned}(19) &= \texttt{! <fun p q -> ~(<}\|e\|\texttt{>)>}\ P\ Q \\ &= \sigma\|e\| \Downarrow^0 \mathbb{Z}\end{aligned}$$

We omit the proof that $\|\texttt{stlcs}\ i\ j\ P\ Q\| \Downarrow^0 \mathbb{Z}$, since it involves no staging. Therefore, by the Erasure Theorem (specifically Lemma 26),

$$\texttt{stlcs}\ i\ j\ P\ Q = \|\texttt{stlcs}\ i\ j\ P\ Q\| \equiv \texttt{lcs}\ i\ j\ P\ Q$$

One can show that $\texttt{lcs}\ i\ j\ P\ Q = \texttt{naive\_lcs}\ i\ j\ P\ Q$, so it follows that

$$\texttt{naive\_lcs}\ i\ j\ P\ Q = \texttt{stlcs}\ i\ j\ P\ Q \qquad \square$$

It's worth noting how the argument let us ignore many details about the generated code. We did track termination and type information, but we never specified what the generated code looks like or what values it should compute. In fact, we were blissfully ignorant of the fact that `stlcs` computes (the length of) the LCS. Erasure thus decouples the reasoning about staging from the reasoning about return values, just as we saw earlier in the `power` example.

In the thesis (Inoue, 2012), it is shown that the proof of $\texttt{naive\_lcs} \approx \texttt{lcs}$ is also quite routine. The lack of surprise in this part of the proof is itself somewhat noteworthy, because it demonstrates that despite the challenges of open-term evaluation (Sections 3 and 5), the impact on correctness proofs is very limited, especially when reasoning about closed, unstaged terms.

6.2 Higher order generators

In this section, we verify a higher order generator—one that takes another code generator as a parameter. The key issue in this scenario is how to specify the behavior of the generators that are passed in as parameters. To this end, we find that proof by careful equalities (Section 4.5) has concrete advantages over proof by normalization.

This section's material is not covered in the thesis (Inoue, 2012). Proof details for this section are given in Appendix A.3.

6.2.1 The code

The code we will analyze is the inlining `fold` function, which captures a very common pattern in which guaranteed inlining can make a significant difference:

```
let rec fold f y xs = match xs with
  | [] -> y
  | x::xs -> fold f (f y x) xs
let stfold f =
  ! <let rec loop y xs = match xs with
       | [] -> y
       | x::xs -> loop ~(f <y> <x>) xs
     in loop>
```

In this listing, `fold` is the usual left-fold function that reduces a list by a binary operator `f`, in a left-associative manner with seed value `y`. The `stfold` function is a staged variant that inlines the binary operator, assuming it is given as a generator

that maps two code values to code. The unstaged and staged functions can be invoked like:

```
fold (+) 0 [1;2;3] (* returns 6 *);;
stfold (fun x y -> <~x + ~y>) 0 [1;2;3] (* returns 6 *);;
```

which both sum together the given list. However, `fold` repeatedly invokes the binary operator for every element whereas `stfold` generates a new loop that inlines the operator, thereby avoiding repeated function calls. In the rest of this section, the symbol f is reserved for the staged binary operator passed in as the first argument to `stfold`.

6.2.2 Correctness proof for CBN

As always, the proof is simpler in CBN than in CBV, so we will present CBN first. We assume *Const* contains integers and lists thereof. Pattern-matches on lists are modeled similarly to `if-then-else`: we include a constant `match`, with the reduction rules:

$$\delta(\texttt{match}, \texttt{[]}) = \lambda g\ h.g\ \texttt{[]}$$
$$\delta(\texttt{match}, c\texttt{::}d) = \lambda g\ h.h\ c\ d.$$

Then, the expression `match xs with [] -> `e_1` | x::xs -> `e_2 is modeled by the λ^U term `match xs (`$\lambda\_.\ e_1$`) (`λ`x xs.`e_2`)`. Ill-formed combinations like δ(`match`,1) are undefined.

We need to be careful about what exactly correctness means for higher order generators like `stfold`. It is not the case that $\forall f, a, l \in \mathit{Arg}.\ \texttt{stfold}\ f\ a\ l \approx \texttt{fold}\ f\ a\ l$, because `stfold` expects f to be a generator whereas `fold` expects f to be an unstaged function. Intuitively, f is a part of the code that `stfold` generates, so it must be erased together with `stfold`. The correct statement to aim for is thus $\forall f, a, l \in \mathit{Arg}.\ \texttt{stfold}\ f\ a\ l \approx \texttt{fold}\ \|f\|\ a\ l$, where f must be a function mapping two code values to a code value.

Proposition 45 (CBN correctness of `stfold`)**.** In CBN, for any $f \in E^0$ such that $\forall x, y \notin \mathrm{FV}(f).\ \exists e.\ \lambda^U_{\mathbf{n}} \vdash f\ \langle y\rangle\ \langle x\rangle = \langle\|e\|\rangle$, we have $\lambda^U_{\mathbf{n}} \vdash \texttt{stfold}\ f = \texttt{fold}\ \|f\|$.

Proof. Using the assumption $f\ \langle y\rangle\ \langle x\rangle = \langle\|e\|\rangle$, we see directly that `stfold` f reduces to an unstaged form, so the Erasure Theorem, specifically Lemma 21, applies. See Appendix A.3.1 for details. □

Example 46. Let $f \stackrel{\text{def}}{\equiv} \lambda x\ y.\langle \tilde{}x + \tilde{}y\rangle$. Then, $f\ \langle y\rangle\ \langle x\rangle = \langle y + x\rangle$ which is of the form $\langle\|e\|\rangle$, so `stfold` f = `fold` $\|f\|$, hence $\lambda^U_{\mathbf{n}} \vdash \texttt{stfold}\ f\ 0 = \texttt{fold}\ \|f\|\ 0 = \texttt{sum}$, where `sum` is a function that sums up all elements of a list.

6.2.3 Correctness proof for CBV using normalization

For CBV, the generated code must additionally terminate to constants under relevant substitutions. In a proof by normalization, this additional condition can be ensured by requiring $f\ \langle y\rangle\ \langle x\rangle$ to produce code that terminates to a constant, as long as

the y and x are substituted by constants drawn from the right domain. For a set $S \subseteq V^0$, let $e \Downarrow^0 S$ mean $\exists v \in S.\ e \Downarrow^0 v$.

Proposition 47 (CBV correctness of `stfold` by normalization)**.** In CBV, let $f \in V^0$ and $D \subseteq \mathit{Const}$, and let $D^* \stackrel{\text{def}}{=} \{[c_1, c_2, \ldots, c_n] \mid c_1, \ldots, c_n \in D,\ n \in \mathbb{N}\}$. Assume that $\forall x, y \notin \mathrm{FV}(f).\ \exists e.\ \lambda^U_{\mathbf{v}} \vdash f\ \langle y \rangle\ \langle x \rangle = \langle \|e\| \rangle$ and, for the same e,

$$\forall c, d \in D.\ \lambda^U_{\mathbf{v}} \vdash \|f\|\ c\ d \Downarrow^0 D \wedge [c, d/y, x]\|e\| \Downarrow^0 D \tag{20}$$

Then, for any $d \in D$ and $l \in D^*$, we have $\lambda^U_{\mathbf{v}} \vdash \texttt{stfold}\ f\ d\ l = \texttt{fold}\ \|f\|\ d\ l$.

Proof. By the Erasure Theorem (specifically, Lemma 26) and some easy arguments, the proof reduces to showing $\texttt{stfold}\ f\ d\ l \Downarrow^0 \mathit{Const}$ and $\|\texttt{stfold}\ f\|\ d\ l \Downarrow^0 \mathit{Const}$. Let $l \equiv [c_1, \ldots, c_n]$. By induction on the length of l, the $\|f\|\ c\ d \Downarrow^0 D$ part of assumption (20) gives a sequence of constants $d_1, \ldots, d_n \in D$ with

$$\|f\|\ d\ c_1 \Downarrow^0 d_1 \qquad \|f\|\ d_1\ c_2 \Downarrow^0 d_2 \qquad \|f\|\ d_2\ c_3 \Downarrow^0 d_3 \qquad \ldots \qquad \|f\|\ d_{n-1}\ c_n \Downarrow^0 d_n$$

and $\|\texttt{stfold}\ f\|\ d\ l \Downarrow^0 d_n$. Similarly, the $[c, d/y, x]\|e\| \Downarrow^0 D$ part of assumption (20) gives another sequence of constants (which need not be proved equal to the d_i's), such that `stfold` $f\ d\ l$ terminates to the last of them. See Appendix A.3.2 for details. □

Example 48. Let $f \stackrel{\text{def}}{\equiv} \lambda x\ y.\langle \tilde{}x + \tilde{}y \rangle$. Then, $\lambda^U_{\mathbf{v}} \vdash f\ \langle y \rangle\ \langle x \rangle = \langle y + x \rangle$, which is of the form $\langle \|e\| \rangle$. Taking $D \stackrel{\text{def}}{=} \mathbb{Z}$, for every $n, m \in \mathbb{Z}$, we have $\|f\|\ n\ m = n + m \Downarrow^0 \mathbb{Z}$ and $[n, m/y, x]\|y + x\| \equiv n + m \Downarrow^0 \mathbb{Z}$, so by Proposition 47, `stfold` $f =$ `fold` $\|f\|$. Hence, `stfold` $f\ 0\ l =$ `fold` $\|f\|\ 0\ l =$ `sum` l in CBV, where l is a list of integers and `sum` is a function that sums up all elements of a list.

Assumption (20) is essentially a type constraint saying that the binary operator f is a first-order function mapping constants to constants. This constraint implicitly gives a set of contexts that force return values to be of ground type (i.e., constants), which is needed to invoke Lemma 26.

6.2.4 Correctness proof for CBV using careful equalities

Assumption (20) in Proposition 47 can be somewhat limiting. For example, it does not cover the case where the binary operator f is itself higher order; for example, f might be (a staged version of) function composition.

We can avoid this kind of restriction by specifying the behavior of f by careful equalities instead. Careful equalities do not dictate the shape of return values, so the statement of correctness becomes cleaner and more general. However, the proof system of Proposition 30 falls short here, because f is invoked only once with a fixed set of arguments whereas the code it produces is invoked multiple times—their reductions simply do not align with each other. The proof must therefore consider the staged and unstaged counterparts separately. Proposition 30 can be used instead for checking the properties of f.

Proposition 49 (CBV correctness of `stfold` by careful equalities)**.** Let $f \in V^0$ and $D \subseteq V^0$, and assume that $\forall x, y \notin \mathrm{FV}(f).\ \exists e.\ \forall u, v \in V^0.\ \lambda^U_{\mathbf{v}\Downarrow}/[v, u/y, x] \vdash f\ \langle y \rangle\ \langle x \rangle = \langle \|e\| \rangle$. Then, $\lambda^U_{\mathbf{v}} \vdash \texttt{stfold}\ f = \texttt{fold}\ \|f\|$.

Proof. By some arguments using extensionality (Proposition 36) and equivalence of divergent terms (Lemma 38), it suffices to show $\forall v_0, l \in V^0.\ \texttt{stfold}\ f\ v_0\ l \approx_{\Uparrow} \|\texttt{stfold}\ f\|\ v_0\ l$. For simplicity, assume $l \stackrel{\text{def}}{\equiv} [u_1, \ldots, u_n]$ and let us focus on the case $\|\texttt{stfold}\ f\|\ v_0\ l \Downarrow^0$. By induction on the length of the input list l, we get a sequence $v_1, \ldots, v_n \in V^0$ such that

$$\|f\|\ v_0\ u_1 = v_1 \qquad \|f\|\ v_1\ u_2 = v_2 \qquad \|f\|\ v_2\ u_3 = v_3 \qquad \ldots \qquad \|f\|\ v_{n-1}\ u_n = v_n$$

and $\|\texttt{stfold}\ f\|\ v_0\ l = v_n$. Starting with v_n and performing right-to-left rewrites using the above equations, one can show that $v_n = \texttt{stfold f}\ v_0\ l$. See Appendix A.3.3 for details. □

Example 50. A function that composes a list of functions can be written as $\texttt{fold}\ (\circ)\ (\lambda z.z)$, where $(\circ) \stackrel{\text{def}}{\equiv} \lambda g\ h\ z.g(h\ z)$ is the composition operator. We'd like to replace $\texttt{fold}\ (\circ)$ by $\texttt{stfold}\ f$, where $f \stackrel{\text{def}}{\equiv} \lambda g\ h.\langle \lambda z.\tilde{}g\ (\tilde{}h\ z)\rangle$, so as to inline the composition operator. It is easy to see that $\lambda^U_{\blacktriangledown\Downarrow}/[v, u/y, x] \vdash f\ \langle y\rangle\ \langle x\rangle = \langle y\ (x\ z)\rangle$ which is of the form $\langle \|e\| \rangle$, so by Proposition 49 we have $\texttt{stfold}\ f\ (\lambda z.z) = \texttt{fold}\ f\ (\lambda z.z)$.

Remark. Technically, this example applies only when composing lists of built-in functions that can be modeled as constants, such as unary `-` or the partial applications of `+` and `*` because lists are modeled as constants. (See Remark 1 about how partially applied operators are viewed as constants.) This restriction can be lifted either by Church-encoding lists or by adding constructors to λ^U, perhaps in the style of Arbiser *et al.* (2006) if the addition of types is undesirable.

6.2.5 Comparison of proofs in CBV

As we have just seen, careful equalities can be better suited for higher order generators than proof by normalization because the former gives a more natural vocabulary for specifying the behavior of the generator coming in as input (f in the case of `stfold`). Then, should we abandon proof by normalization and always work with careful equalities? Not necessarily, as one can check in Appendices A.3.2 and A.3.3, once the restriction to first-order is in place, the details of the proof can be much simpler in the normalization approach. Only when the restriction to first-order is unacceptable do we need careful equalities.

Note that this limitation to first-order is not as grave as it may seem. If f can be given a fixed type, the returned code can be applied to more arguments to force ground-type return values. In Example 50, if the input list is known to contain only functions of type `int -> int`, a correctness proof by normalization simply needs to prove $\texttt{stfold}\ f\ m\ l\ k = \texttt{fold}\ (\circ)\ m\ l\ k$, with an extra argument $k \in \mathbb{Z}$, so that the return type is `int`. Moreover, higher orderness is itself an abstraction that ought to be eliminated with staging, so practical generators tend to produce first-order code. As a case in point, the code generation in Example 50 is barely beneficial in practice, since performance gains from inlining $(\circ)$ are dwarfed by the costs of repeatedly allocating the results of the inlined $(\circ)$.

An important class of generators that genuinely require higher order correctness statements is staged interpreters for higher order languages (Brady & Hammond,

2006; Taha, 2008; Carette *et al.*, 2009). When a programming language interpreter is written in MetaOCaml and staged, the result is a translator—i.e., a compiler—from the object language to OCaml. If the object language has higher order features, the generated code may have to be higher order as well (though that can limit the gains from staging). Hence, it is desirable to have an alternative correctness proof that allows the generated code to be higher order. In the absence of such a functional requirement, proof by normalization can be a sensible choice.

7 Related works

Taha (1999) first discovered λ^U, which showed that functional hygienic MSP admits intensional equalities like β, even under brackets. The key was to drop intentional analysis, or pattern-matching on the syntactic structure of code values. By contrast, earlier systems that allowed intentional analysis were forced to have trivial equational theories (Muller, 1992). However, Taha showed the mere existence of the theory and did not explore how to use it for verification or investigate extensional equivalences. Moreover, though Taha laid down the operational semantics of both CBV and CBN, he gave an equational theory for only CBN and left the trickier CBV unaddressed.

Yang (2000) pioneered the use of an "annotation erasure theorem", which stated $e \Downarrow^0 \langle \|t\| \rangle \Longrightarrow \|t\| \approx \|e\|$. But there was a catch: The conclusion $\|t\| \approx \|e\|$ was asserted in the unstaged base language, instead of the staged language. Translated to our setting, the conclusion of the theorem was $\lambda \vdash \|t\| \approx \|e\|$ and not $\lambda^U \vdash \|t\| \approx \|e\|$. In practical terms, this meant that the context of deployment of the staged code could contain no further staging. Code generation must be done offline, and application programs using the generated $\|t\|$ must be written in a single-stage language, or else no guarantee was made. This interferes with combining analyses of multiple generators and precludes dynamic code generation by run (!). Yang also worked with operational semantics and did not explore in depth how equational reasoning interacts with erasure.

This paper can be seen as a confluence of these two lines of research: we complete λ^U by giving a CBV theory with a comprehensive study of its peculiarities, and adapt erasure to produce an equality in the staged language λ^U.

Berger & Tratt (2015) devised a Hoare-style program logic for the typed language Mini-ML$_e^{\Box}$. They develop a promising foundation and prove strong properties about it, such as relative completeness, but concrete verification tasks they consider concern relatively simple programs. Mini-ML$_e^{\Box}$ also prohibits manipulating open terms, so it does not capture the difficulty of reasoning about free variables, which is one of the main challenges we face up to. Insights gained from λ^U should help extend such logics to more expressive languages, and our proof techniques will be a good toolbox to lay on top of them.

An interesting line of work that mitigates the expressivity problems in Mini-ML$_e^{\Box}$ yet successfully avoids issues with open terms is contextual modal type theory (Nanevski *et al.*, 2008). Its application to MSP offers a staging construct which, through typing, restricts code values to closed terms of a form that roughly translates

to

$$\lambda x_1. \ldots \lambda x_n. \langle [\tilde{}x_1, \ldots, \tilde{}x_n / x_1, \ldots, x_n] e^0 \rangle$$

in λ^U notation. That is, code values must be closed, and any references to level-1 (or higher) free variables must be expressed via reference to escaped level-0 variables. The resulting closure is applied to terms like $\langle x \rangle$, replacing the escaped level-0 variables by level-1 variables. (Strictly speaking, both the abstraction and the application to $\langle x \rangle$ use custom constructs, so the open term $\langle x \rangle$ is never explicitly constructed as a first-class value.) We expect the Erasure Theorem to still apply to this setting and be augmented with cleaner characterizations of observational equivalence than those developed in this paper.

For MSP with variable capture, Choi *et al.* (2011) proposed an alternative approach with different trade-offs than ours. They provide an "unstaging" translation of staging annotations into environment-passing code. Their translation is semantics-preserving with no proof obligations but leaves an unstaged program that is complicated by environment-passing, whereas our erasure approach leaves a simpler unstaged program at the expense of additional proof obligations. Their approach also has the advantage that the target language of the translation has no staging, so reasoning principles need not be ported to that setting—provided that, like with Yang's results, the context of deployment contains no further staging. It will be interesting to see how these approaches compare in practice or if they can be usefully combined, but, for the moment, they seem to fill different niches.

There is a wealth of publications on representing free variables and binding structures, often with the goal of supporting syntactic transformations and/or mechanized reasoning (Gabbay & Pitts, 2001; Aydemir *et al.*, 2008; Licata *et al.*, 2008; Pouillard & Pottier, 2010). The nominal (Gabbay & Pitts, 2001) and definitional variation (Licata *et al.*, 2008) approaches in particular provide deep insights into the mathematical properties of binding and scope. While the present paper gives only an operational intuition as to the cause of pathologies relating to open-term manipulation (see Section 3), these more developed theories of binding may be able to provide more formal, mathematical explanations.

Applicative bisimulation has been studied extensively as a characterization of observational equivalence in the plain λ calculus and its variants (Abramsky, 1990; Howe, 1996; Gordon, 1999), which made it a natural starting point in our investigation. However, more advanced flavors of bisimulation exist, offering greater flexibility and lighter proof obligations. Small bisimulations (Koutavas & Wand, 2006) and environmental bisimulations (Sangiorgi *et al.*, 2011) do not directly relate terms but more abstract states that can track contextual information, allowing the handling of effects. It will be interesting to see how they can be adapted to the multi-stage setting. Up-to techniques are often indispensable in simplifying the proof obligations for establishing bisimilarity between concrete terms (Pous & Sangiorgi, 2011). Our Definition 32 builds in reasoning up-to observational equivalence, which can be seen as a reformulation of bisimulation-up-to-bisimilarity (Milner, 1989) with the understanding that bisimilarity coincides with observational equivalence. This enhancement simplified the proof of extensionality (Proposition 36).

There are also other characterizations of observational equivalence. The CIU Theorem (Mason & Talcott, 1991) states that terms are observationally equivalent iff all of their closed instances equiterminate under arbitrary evaluation contexts. The Context Lemma (Milner, 1977; Jim & Meyer, 1996) states, in a typed setting, that closed terms of some type τ are equivalent exactly when they cannot be distinguished by the elimination forms for the type τ. Both approaches reduce the set of contexts that must be considered and are arguably simpler than bisimulation. As a result, Mason and Talcott observe that the proofs tend to be simpler (Mason & Talcott, 1991). We have not investigated how these techniques can be adapted to λ^U.

8 Conclusion and future work

We have addressed three basic concerns for verifying staged programs. First, we showed that staging is a non-conservative extension because reasoning under substitutions is unsound in a multi-stage language, even if we are dealing with unstaged terms. Despite this drawback, untyped functional MSP has a rich set of useful properties. Second, we proved that simple termination conditions guarantee that erasure preserves semantics, which reduces the problem of proving the irrelevance of annotations on a program's semantics to the better studied problem of proving termination. Finally, we showed a sound and complete notion of applicative bisimulation in this setting, which allows us to reason under substitution in some cases. In particular, the shocking lack of β_x in $\lambda^U_{\mathbf{v}}$ is of limited practical relevance as we have $C\beta_x$ instead, which covers β_x completely when we are dealing with closed, erased terms.

These results yield important insights into the semantics of hygienic MSP. The Erasure Theorem gives intuitions on what staging annotations can or cannot do, with which we may educate the novice multi-stage programmer. Applicative bisimulation adapts in a natural manner and the familiar notion of function extensionality carries over. The key difference from single-stage languages is the behavior of free variables, which greatly affect the formulation of bisimulation. However, the notion of bisimulation that we formulated in light of this difference is sound and complete, suggesting that free variables' behavior is the only essential difference between λ^U and λ. This broad set of insights has brought us to a level where the correctness proof of a sophisticated generator like LCS is easily within reach, as are similar proofs for higher order generators.

This work may be extended in several interesting directions. We have specifically identified some open questions about λ^U: Which type systems, if any, allow reasoning under substitutions? Is λ^U conservative over the plain λ calculus for closed terms? Can the extensionality principle be strengthened to require equivalence for only closed-term arguments? What is a sensible notion of partial erasure, and is it useful for stating and proving correctness of higher order generators? Answering these questions will strengthen our understanding of staging even further.

In this paper, we pointed out that λ^U is not conservative in the sense that not all observationally equivalent terms in the standard CBN and CBV λ calculi

(Plotkin, 1975) remain equivalent in λ^U. However, a major theme in this paper is that λ^U nonetheless conserves useful reasoning principles. The $\beta/\beta_{\mathbf{v}}$ equality with confluence, β_x, extensionality, and sound and complete applicative bisimulation all carry over, albeit with some changes, from the standard λ calculi. For $\beta/\beta_{\mathbf{v}}$, we have also established that the changes—namely, level restrictions—cannot be reduced any further. For applicative bisimulation, its completeness suggests that the modifications from the plain λ calculus are minimal. It will be very interesting to explore if other reasoning principles, like recursion induction (McCarthy, 1963), carry over, and how much modification is strictly necessary.

It will also be interesting to investigate which of the more advanced alternatives to applicative bisimulation (Milner, 1977; Mason & Talcott, 1991; Koutavas & Wand, 2006; Sangiorgi *et al*., 2011) can be adapted to the multi-stage setting. Many of them have had success in handling effects, so they may make imperative hygienic MSP languages (Westbrook *et al*., 2010; Kameyama *et al*., 2011; Rompf & Odersky, 2012) susceptible to analysis. However, like with applicative bisimulation, it seems common practice in these techniques to assume that only closed instances of terms matter. Koutavas and Wand, for instance, start by ruling out open terms in the states being compared. Thus, these techniques will probably need similar treatment to applicative bisimulation in order to track substitutions. Perhaps, environmental bisimulation can capture them with little modification, using its existing machinery for tracking contextual information.

As a caveat, the Erasure Theorem does not apply as-is to imperative languages, since modifying evaluation strategies can commute the order of effects. Two mechanisms will be key in studying erasure for imperative languages—one for tracking which effects are commuted with which, and another for tracking mutual (in)dependence of effects, perhaps separation logic (Reynolds, 2002) for the latter. In any case, investigation of imperative hygienic MSP may have to wait until the foundation matures, as noted in the introduction. Adapting erasure and other techniques to lightweight modular staging (Rompf & Odersky, 2012), like we noted in the introduction, will need further development. The additional challenge there is to cope with the flexibility in the semantics that can be attached to the object code. It may require the host language semantics to be able to mix different semantics, so that the erasure makes sense.

Devising a mechanized program logic would also be an excellent goal. Berger and Tratt's program logic (2015) may be a good starting point, although whether to go with Hoare logic or to recast it in equational style is an interesting design question. A mechanized program logic may let us automate the particularly MSP-specific proof step of showing that erasure preserves semantics. The Erasure Theorem reduces this problem to essentially termination checks, and we can probably capitalize on recent advances in automated termination analysis, for example, those of Heizmann *et al.* (2010).

Finally, this work focused on functional (input–output) correctness of staged code, but quantifying performance benefits is also an important concern for a staged program. It will be interesting to see how we can quantify the performance of a staged program through formalisms like improvement theory (Sands, 1998).

Acknowledgments

We would like to thank Gregory Malecha, Edwin Westbrook, and Mathias Ricken for their input on the technical content of this paper. The higher order generator example was motivated by discussions with Yasuhiko Minamide and Yukiyoshi Kameyama. We are grateful to Veronica Gaspez, Bertil Svensson, and the anonymous reviewers for their feedback. The first author has received significant guidance and supervision from Robert Cartwright, Vivek Sarkar, and Marcia O'Malley, who, together with the second author, constituted the first author's doctoral thesis committee. We thank Carol Sonenklar, Ray Hardesty, and Mark Stephens for their input on the writing.

References

Abramsky, S. (1990) The lazy lambda calculus. In *Research Topics in Functional Programming*, Turner, D. A. (ed). Addison-Wesley, pp. 65–116.

Arbiser, A., Miquel, A. & Ríos, A. (2006) A lambda-calculus with constructors. In Proceedings of Term Rewriting and Applications, 17th International Conference. LNCS, vol. 4098. Springer Berlin Heidelberg, pp. 181–196.

Aydemir, B., Charguéraud, A., Pierce, B. C., Pollack, R. & Weirich, S. (2008) Engineering formal metatheory. *SIGPLAN Not.* **43**(1), 3–15.

Barendregt, H. P. (1984) *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and The Foundations of Mathematics. Amsterdam, New York, Oxford: North-Holland Publishing Company.

Benaissa, Z. El-abidine, Moggi, E., Taha, W. & Sheard, T. (1999) Logical modalities and multi-stage programming. In Federated Logic Conference (FLoC) Satellite Workshop on Intuitionistic Modal Logics and Applications (IMLA).

Berger, M. & Tratt, L. (2015) Program logics for homogeneous generative run-time meta-programming. *Log. Meth. Comput. Sci.* **11**(1), 1–50.

Bondorf, A. (1992) Improving binding times without explicit CPS-conversion. In Proceedings of the 1992 ACM Conference on LISP and Functional Programming. New York, NY, USA: ACM, pp. 1–10.

Brady, E. & Hammond, K. (2006) A verified staged interpreter is a verified compiler. In Proceedings of Generative Programming and Component Engineering, 5th International Conference (GPCE). New York, NY, USA: ACM, pp. 111–120.

Carette, J. & Kiselyov, O. (2011) Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Sci. Comput. Program.* **76**(5), 349–375.

Carette, J., Kiselyov, O. & Shan, C.-C. (2009) Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* **19**(5), 509–543.

Choi, W., Aktemur, B., Yi, K. & Tatsuta, M. (2011) Static analysis of multi-staged programs via unstaging translation. In Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). New York, NY, USA: ACM, pp. 81–92.

Cohen, A., Donadio, S., Garzaran, M.-J., Herrmann, C., Kiselyov, O. & Padua, D. (2006) In search of a program generator to implement generic transformations for high-performance computing. *Sci. Comput. Program.* **62**(1), 25–46.

Dybvig, R. K. (1992) *Writing Hygienic Macros in Scheme with Syntax-Case*. Technical Reports TR356. Indiana University Computer Science Department.

Gabbay, J. M. & Pitts, M. A. (2001) A new approach to abstract syntax with variable binding. *Formal Asp. Comput.* **13**(3), 341–363.

Gordon, A. D. (1999) Bisimilarity as a theory of functional programming. *Theor. Comput. Sci.* **228**(1–2), 5–47.

Heizmann, M., Jones, N. D. & Podelski, A. (2010) Size-change termination and transition invariants. In Proceedings of Static Analysis - 17th International Symposium (SAS). LNCS, vol. 6337. Springer Berlin Heidelberg, pp. 22–50.

Herrmann, C. A., & Langhammer, T. (2006) Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. *Sci. Comput. Program.* **62**(1), 47–65.

Howe, D. J. (1996) Proving congruence of bisimulation in functional programming languages. *Inf. Comput.* **124**(2), 103–112.

Inoue, J. (2012) *Reasoning about Multi-Stage Programs*. Ph.D. thesis, Rice University.

Inoue, J. & Taha, W. (2012) Reasoning about multi-stage programs. In Proceedings of the 21st European Symposium on Programming (ESOP). LNCS, vol. 7211. Springer Berlin Heidelberg, pp. 357–376.

Intrigila, B. & Statman, R. (2009) The omega rule is Π_1^1-complete in the $\lambda\beta$-calculus. *Log. Meth. Comput. Sci.* **5**(2), 1–21.

Jim, T. & Meyer, A. R. (1996) Full abstraction and the context lemma. *SIAM J. Comput.* **25**(3), 663–696.

Kameyama, Y., Kiselyov, O. & Shan, C.-C. (2011) Shifting the stage - staging with delimited control. *J. Funct. Program.* **21**(6), 617–662.

Kameyama, Y., Kiselyov, O. & Shan, C.-C. (2015) Combinators for impure yet hygienic code generation. *Sci. Comput. Program.* **112**(P2), 120–144.

Kim, Ik-S., Yi, K. & Calcagno, C. (2006) A polymorphic modal type system for LISP-like multi-staged languages. In Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). New York, NY, USA: ACM, pp. 257–268.

Kiselyov, O. & Taha, W. (2005) Relating FFTW and split-radix. In Proceedings of Embedded Software and Systems, First International Conference (ICESS). LNCS, vol. 3605. Springer Berlin Heidelberg, pp. 488–493.

Koutavas, V. & Wand, M. (2006) Small bisimulations for reasoning about higher-order imperative programs. In Proceedings of 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. New York, NY, USA: ACM, pp. 141–152.

Licata, D. R., Zeilberger, N. & Harper, R. (2008) Focusing on binding and computation. In Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science. LICS '08. Washington, DC, USA: IEEE Computer Society, pp. 241–252.

Mason, I. & Talcott, C. (1991) Equivalence in functional languages with effects. *J. Funct. Program.* **1**(7), 287–327.

McCarthy, J. (1963) A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, Braffort, P. & Hirschberg, D. (eds). North-Holland, pp. 33–70.

McCusker, G. (2003) On the semantics of the bad-variable constructor in Algol-like languages. *Electron. notes Theor. Comput. Sci.* **83**, 169–186.

Milner, R. (1977) Fully abstract models of the typed lambda-calculus. *Theor. Comput. Sci.* **4**(1), 1–22.

Milner, R. (1989) *Communication and Concurrency*. Prentice-Hall.

Mitchell, J. C. (1993) On abstraction and the expressive power of programming languages. *Sci. Comput. Program.* **21**(2), 141–163.

Muller, R. (1992) M-LISP: A representation-independent dialect of LISP with reduction semantics. *ACM Trans. Program. Lang. Syst.* **14**(4), 589–616.

Nanevski, A., Pfenning, F. & Pientka, B. (2008) Contextual modal type theory. *ACM Trans. Comput. Logic* **9**(3), 23:1–23:49.

Plotkin, G. D. (1974) The λ-calculus is ω-incomplete. *J. Symb. Logic* **39**(2), 313–317.

Plotkin, G. D. (1975) Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci.* **1**(2), 125–159.

Pouillard, N. & Pottier, F. (2010) A fresh look at programming with names and binders. In Proceedings of the 15th ACM Sigplan International Conference on Functional Programming. ICFP '10. New York, NY, USA: ACM, pp. 217–228.

Pous, D. & Sangiorgi, D. (2011) Enhancements of the bisimulation proof method. In *Advanced Topics in Bisimulation and Coinduction*, Sangiorgi, D. & Rutten, J. (eds). Cambridge University Press. Cambridge Books Online, pp. 233–289.

Reynolds, J. C. (2002) Separation logic: A logic for shared mutable data structures. In Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS). Washington, DC, USA: IEEE Computer Society, pp. 55–74.

Riecke, J. G. & Subrahmanyam, R. (1994) Extensions to type systems can preserve operational equivalences. In Proceedings of Theoretical Aspects of Computer Software (TACS). LNCS, vol. 789. Springer Berlin Heidelberg, pp. 76–95.

Rompf, T. & Odersky, M. (2012) Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* **55**(6), 121–130.

Sands, D. (1998) Improvement theory and its applications. In *Higher Order Operational Techniques in Semantics*, Gordon, A. D. & Pitts, A. M. (eds). Cambridge University Press, pp. 275–306.

Sangiorgi, D., Kobayashi, N. & Sumii, E. (2011) Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.* **33**(1), 5:1–5:69.

Swadi, K., Taha, W., Kiselyov, O. & Pašalić, E. (2006) A monadic approach for avoiding code duplication when staging memoized functions. In Proceedings of Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM). New York, NY, USA: ACM, pp. 160–169.

Taha, W. (1999) *Multistage Programming: Its Theory and Applications*. Ph.D. thesis, Oregon Graduate Institute.

Taha, W. (2008) A gentle introduction to multi-stage programming, part II. In Proceedings of Generative and Transformational Techniques in Software Engineering II. LNCS, vol. 5235. Springer Berlin Heidelberg, pp. 260–290.

Taha, W. & Nielsen, M. F. (2003) Environment classifiers. In Proceedings of the 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). New York, NY, USA: ACM, pp. 26–37.

Takahashi, M. (1995) Parallel reductions in lambda-calculus. *Inf. Comput.* **118**(1), 120–127.

Tsukada, T. & Igarashi, A. (2010) A logical foundation for environment classifiers. *Log. Meth. Comput. Sci.* **6**(4), 1–43.

Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T. & Taha, W. (2010) Mint: Java multi-stage programming using weak separability. In Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). New York, NY, USA: ACM, pp. 400–411.

Yang, Z. (2000) *Reasoning About Code-Generation in Two-Level Languages*. Technical Report RS-00-46. BRICS.

Yuse, Y., & Igarashi, A. (2006) A modal type system for multi-level generating extensions with persistent code. In Proceedings of the 8th ACM SIGPLAN International Conference

on Principles and Practice of Declarative Programming (PPDP). New York, NY, USA: ACM, pp. 201–212.

Appendix A. Proofs

This appendix includes formalizations of claims and proofs of theorems that were omitted from the main text. The proofs are kept brief, just enough to get the idea across. For complete details, see the thesis (Inoue, 2012).

A.1 Equivalence of open- and closed-term observations

In this section, we prove that Definition 6 (observational equivalence) is equivalent to the stratified definition with non-closing contexts used by Taha (1999). We recall Taha's definition first. We will ignore constants throughout this section, but adding them is straightforward.

Definition 51 (Stratified, non-closing observational equivalence)**.** For each ℓ, define $e \approx_\ell t$ iff $e, t \in E^\ell$ and for every C such that $C[e], C[t] \in E^0$, we have $C[e] \Downarrow^0 \Longleftrightarrow C[t] \Downarrow^0$.

The idea behind the proof is that observation of open terms can be recast as observation of closed terms. In CBV, the machinery to do this recasting is λ^U's ability to force evaluations of open terms within programs (which are closed by definition). In CBN, the machinery does not rely on staging and requires a lemma that also holds in the plain λ calculus.

Lemma 52. Let $\sigma, \sigma' : Var \xrightarrow[\text{fin}]{} \{t \in E^0 : t \Uparrow^0\}$ be substitutions that substitute only divergent level-0 terms. Then, $\sigma e \Downarrow^\ell \Longleftrightarrow \sigma' e \Downarrow^\ell$ for any ℓ, e.

Proof. By symmetry, proving $\sigma e \Downarrow^\ell \Longrightarrow \sigma' e \Downarrow^\ell$ will suffice. The proof proceeds by induction on the number of steps that σe takes to terminate, using a technical lemma to classify the shape of σe. See the thesis for details. □

Remark. Note that it need *not* be the case that $\operatorname{dom}\sigma \supseteq \mathrm{FV}(e)$.

Proposition 53. $(\approx) = \bigcup_\ell (\approx_\ell)$.

Proof. Suppose $(e, t) \in \bigcup_\ell (\approx_\ell)$. Then, for some ℓ, we have $e, t \in E^\ell$ and $e \approx_\ell t$. Then, every context C with $C[e], C[t] \in Prog$ also satisfies $C[e], C[t] \in E^0$, so $C[e]$ and $C[t]$ equi-terminate. Therefore, $e \approx t$, which shows $\bigcup_\ell (\approx_\ell) \subseteq (\approx)$.

For the converse, suppose $e \not\approx_\ell t$. Then, for some C we have $C[e], C[t] \in E^0$ but $C[e]$ and $C[t]$ do not equiterminate. Recalling that we are omitting the treatment of constants, without loss of generality we have $C[e] \Uparrow^0$ but $C[t] \Downarrow^0$. Let $\lambda\overline{x_i}$ denote a sequence of λ's that bind all free variables in $C[e]$ and $C[t]$.

In CBV, let $e_1 ; e_2$ denote sequencing, which checks that e_1 terminates, discarding the return value, and then evaluates e_2. Sequencing is just syntactic sugar for $(\lambda\_.e_2)\, e_1$ in CBV. Then, the context $C' \stackrel{\text{def}}{\equiv} \langle \lambda\overline{x_i}.\tilde{}(C ; \langle 0 \rangle) \rangle$ satisfies $C'[e], C'[t] \in Prog$, yet $C'[e] \Uparrow^0$ and $C'[t] \Downarrow^0$, so $\lambda^U_{\mathbf{v}} \vdash e \not\approx t$.

In CBN, let Ω be a closed, divergent level-0 term. Then, $C' \stackrel{\text{def}}{\equiv} (\lambda\overline{x_i}.C)\ \overline{\Omega}$ is a program context for e and t, where there are as many copies of Ω as there are

variables in $\overline{x_i}$. Now, $C[e] \Uparrow^0 \Longleftrightarrow C'[e] \Uparrow^0$ and $C[t] \Downarrow^0 \Longleftrightarrow C'[t] \Downarrow^0$ by Lemma 52 and SS-β, so $\lambda_{\mathbf{n}}^U \vdash e \not\approx t$. □

A.2 Soundness and completeness of applicative bisimulation

This section explains the proof of Theorem 35. We basically just adapt Howe's method (1996), but the details are complicated by the inconsistent handling of substitutions in λ^U's bisimulation. Definition 34 says that terms must be compared under all substitutions, yet Definition 32, for $(\lambda x.e)\ R_\dagger^0\ (\lambda x.t)$, says that only the substitutions that eliminate x should matter. When we try to prove Theorem 35 by coinduction, we find that Definition 32 refers not to the bisimulation whose definition it is a part of, but to a different bisimulation that holds only under substitutions that eliminate x, undermining the coinduction. To solve this problem, we recast bisimulation to a family of relations indexed by a set of variables to be eliminated, so that the analogue of Definition 32 can refer to a different member of the family. Theorem 35 is then proved by mutual coinduction.

A.2.1 Overview

We first review Howe's method for single-stage calculi and motivate the change that we made in adapting it to λ^U, focusing on CBV. Howe shows that bisimilarity $(\sim)$, the union of all bisimulations, is a non-trivial congruence, whence $(\sim) = (\approx)$. The hardest part is showing that $(\sim)$ respects contexts, i.e., $e \sim t \Longrightarrow C[e] \sim C[t]$. For this step, Howe defines an auxiliary relation $e \mathrel{\widehat{\sim}} t$, the *precongruence candidate*, which holds iff e can be transformed into t by one bottom-up pass of replacing successively larger subterms e' of e by some t' such that $e' \sim t'$. Formally, this relation is defined along the following lines:

$$\frac{x \sim t}{x \mathrel{\widehat{\sim}} t} \qquad \frac{c \sim t}{c \mathrel{\widehat{\sim}} t} \qquad \frac{e \mathrel{\widehat{\sim}} s \qquad \lambda x.s \sim t}{\lambda x.e \mathrel{\widehat{\sim}} t} \qquad \frac{e_1 \mathrel{\widehat{\sim}} s_1 \qquad e_2 \mathrel{\widehat{\sim}} s_2 \qquad s_1\ s_2 \sim t}{e_1\ e_2 \mathrel{\widehat{\sim}} t} \qquad \text{etc.}$$

where s ranges over E. Clearly, $(\sim) \subseteq (\mathrel{\widehat{\sim}})$ holds and $(\mathrel{\widehat{\sim}})$ respects contexts. Howe shows that $(\mathrel{\widehat{\sim}})$ is also a bisimulation and concludes that $(\mathrel{\widehat{\sim}}) = (\sim)$ because $(\sim)$ contains all bisimulations. Therefore, $(\sim)$ respects contexts.

If we try to apply this idea to λ^U directly, we get stuck in the proof that $(\mathrel{\widehat{\sim}})$ is a bisimulation. Concretely, we cannot seem to prove

$$\forall e, t \in E^0.\quad \lambda x.e \mathrel{\widehat{\sim}}_\dagger^0 \lambda x.t \Longrightarrow \lambda x.e \mathrel{\widehat{\sim}} \lambda x.t \tag{A 1}$$

Implication (A 1) arises as a subgoal in an inductive proof that $e' \mathrel{\widehat{\sim}} t'$ and $e' \Downarrow^\ell u$ imply $t' \Downarrow^\ell v$ and $u \mathrel{\widehat{\sim}}_\dagger^\ell v$, where $\ell = \max(\mathrm{lv}\, e, \mathrm{lv}\, t)$. The induction is on the number of steps e' takes to terminate. Recall that we are focusing on CBV and consider the case

- $\ell = 0$,
- $e' \equiv (\lambda y.e_1)\ e_2 \wedge t' \equiv (\lambda y.t_1)\ t_2$, and
- $e_2 \Downarrow^0 \lambda x.e \wedge t_2 \Downarrow^0 \lambda x.t$.

By inductive hypothesis, we get $\lambda x.e \mathrel{\widehat{\sim}}_\dagger^0 \lambda x.t$, the antecedent of implication (A 1). We would like to show $[\lambda x.e/y]e_1 \mathrel{\widehat{\sim}} [\lambda x.t/y]t_1$ holds: Because $[\lambda x.e/y]e_1$ terminates

in fewer steps than $(\lambda y.e_1)\ e_2$, the inductive hypothesis would immediately show that $[\lambda x.t/y]t_1$ terminates to a suitable value. However, showing $[\lambda x.e/y]e_1 \mathrel{\widehat{\sim}} [\lambda x.t/y]t_1$ for general e_1 and t_1 entails showing $\lambda x.e \mathrel{\widehat{\sim}} \lambda x.t$, leaving us with the goal (A 1).

Unfortunately, trying to prove assertion (A 1) directly seems rather hopeless because it involves showing a kind of function extensionality result that we want to show *using* bisimulation (cf. Proposition 36) and not the other way around. More specifically, the antecedent of formula (A 1) is equivalent to $\forall a.\ [a/x]e \mathrel{\widehat{\sim}} [a/x]t$, but this cannot imply $e \mathrel{\widehat{\sim}} t$ if we are to have $(\widehat{\sim}) = (\sim) = (\approx)$, since reasoning about $(\approx)$ under substitutions is unsound. Thus, our only chance of proving assertion (A 1) is to somehow show $\lambda x.e \mathrel{\widehat{\sim}} \lambda x.t$ directly on the basis of $\forall a.\ [a/x]e \mathrel{\widehat{\sim}} [a/x]t$, which is just function extensionality. Defining $R^0_\dagger$ to compare function bodies under substitutions was a vital provision for specific applications like Proposition 36 and for making bisimulation complete with respect to $(\approx)$ in general; however, in the soundness proof it comes up as a significant roadblock.

In Howe's setting, which prohibits open-term evaluation, this problem does not arise because everything is compared under closing substitutions. He defines $e' \mathrel{\widehat{\sim}} t'$ on open terms to hold iff $\sigma e'$ and $\sigma t'$ satisfy certain conditions for every closing σ, so the conditional assertion $\forall a.\ [a/x]e \mathrel{\widehat{\sim}} [a/x]t$ that only assures $(\widehat{\sim})$ under $[a/x]$ coincides with $e \mathrel{\widehat{\sim}} t$. In such a setting, defining $\lambda x.e \mathrel{\widehat{\sim}}_\dagger \lambda x.t$ as $e \mathrel{\widehat{\sim}} t$ works fine, making Equation (A 1) trivial, whereas in λ^U this is unsatisfactory because $(\forall a.\ [a/x]e \approx [a/x]t) \not\Longrightarrow e \approx t$.

To solve this problem, we generalize bisimilarity to a family of relations $e \sim_X t$ indexed by a set of variables X, which hold iff $\sigma e \sim \sigma t$ under all substitutions $\sigma : Var \xrightarrow[\text{fin}]{} Arg$ with $\operatorname{dom}\sigma \supseteq X$. As explained in Section 5.1, this has the effect that in CBV we can assume the variables in X to have been substituted with values, though in CBN it has no effect. Then, relations under experiment are refined, essentially, to $\lambda x.e \mathrel{\widehat{\sim}}^0_\dagger \lambda x.t \iff e \mathrel{\widehat{\sim}}_{\{x\}} t$, and $\lambda x.e \mathrel{\widehat{\sim}}_X \lambda x.t$ is refined to

$$\frac{e^0 \mathrel{\widehat{\sim}}_X s \qquad \lambda x.s \sim_{X\setminus\{x\}} t}{\lambda x.e^0 \mathrel{\widehat{\sim}}_{X\setminus\{x\}} t}$$

Then, the family $(\widehat{\sim}_X)$ respects contexts with diminishing indices, i.e., $e \mathrel{\widehat{\sim}}_X t \Longrightarrow \forall C.\ \exists Y \subseteq X.\ C[e] \mathrel{\widehat{\sim}}_Y C[t]$. In particular, $\lambda x.e \mathrel{\widehat{\sim}}^0_\dagger \lambda x.t \iff e \mathrel{\widehat{\sim}}_{\{x\}} t \Longrightarrow \lambda x.e \mathrel{\widehat{\sim}}_\varnothing \lambda x.t$, so we have an analogue of assertion (A 1) with indices added. With this modification, the rest of the proof goes smoothly.

A.2.2 The proof

We now move on to a formal presentation of the proof. The following applies to both CBV and CBN. To simplify the proof, we will mostly use observational order rather than observational equivalence.

Definition 54 (Observational order)**.** Define $e \lessapprox t$ iff for every C such that $C[e], C[t] \in Prog$, $C[e] \Downarrow^0 \Longrightarrow C[t] \Downarrow^0$ holds and whenever $C[e]$ terminates to a constant, then $C[t]$ terminates to the same constant.

Remark. Note $(\approx) = (\lessapprox) \cap (\gtrapprox)$.

Notation. A sequence or family of mathematical objects x_i indexed by a set I is written $\overline{x_i}^{i\in I}$. The superscript binds the index variable i. The superscript may be abbreviated like $\overline{x_i}^{i}$ or omitted if the intention is clear. Let X, Y range over $\wp_{\text{fin}} Var$, the set of all finite subsets of Var. Let $\overline{R_X}$ denote a family of relations R_X indexed by $X \in \wp_{\text{fin}} Var$. Union, inclusion, and other operators between families work pointwise, e.g., $\overline{R_X} \subseteq \overline{S_X}$ denotes $\forall X.\ R_X \subseteq S_X$. Let the signature $\sigma : X | Var \underset{\text{fin}}{\rightharpoonup} Arg$ mean that $\sigma : Var \underset{\text{fin}}{\rightharpoonup} Arg$ and $\operatorname{dom}\sigma \supseteq X$, i.e., σ substitutes for at least the variables in X. For a relation R, let R^{-1} denote $\{(e,t) \mid tRe\}$.

Definition 55 (Indexed relation under experiment)**.** Given a family of relations $\overline{R_X}^X$ with $R_X \subseteq E \times E$ for each X, define $\{\overline{R_X}\}^\ell \subseteq V^\ell \times V^\ell$ for each level ℓ by

$$\frac{}{c\ \{\overline{R_X}\}^0\ c} \qquad \frac{e^0\ R_{\{x\}}\ t^0}{(\lambda x.e^0)\ \{\overline{R_X}\}^0\ (\lambda x.t^0)} \qquad \frac{e^0\ R_\emptyset\ t^0}{\langle e^0\rangle\ \{\overline{R_X}\}^0\ \langle t^0\rangle} \qquad \frac{u^{\ell+1}\ R_\emptyset\ v^{\ell+1}}{u^{\ell+1}\ \{\overline{R_X}\}^{\ell+1}\ v^{\ell+1}}$$

Definition 56 (Indexed applicative bisimilarity)**.** Given a family of relations $\overline{R_X}^X$, define a family of relations $\overline{[R]_X}^X$ by

$$\begin{aligned} e\,[R]_X\,t \overset{\text{def}}{\iff}\ & \forall \sigma : X | Var \underset{\text{fin}}{\rightharpoonup} Arg.\ \text{let } \ell = \max(\operatorname{lv} e, \operatorname{lv} t) \text{ in} \\ & (\sigma e \Downarrow^\ell \Longrightarrow \sigma t \Downarrow^\ell) \wedge (\sigma e \Downarrow^\ell u \wedge \sigma t \Downarrow^\ell v \Longrightarrow u\ \{\overline{R_X}\}^\ell\ v) \end{aligned} \tag{A 2}$$

Indexed applicative similarity $\overline{(\lesssim_X)}$ is the largest family of relations such that $\overline{(\lesssim_X)} = \overline{[\lesssim]_X}$, i.e., it is defined by replacing R by $\lesssim$ in Equation (A 2) and reading the result coinductively. *Indexed applicative bisimilarity* $\overline{(\sim_X)}$ is the largest family of relations such that $\overline{(\sim_X)} = \overline{[\sim]_X}^X \cap \overline{[(\sim)^{-1}]_X}^{-1X}$, i.e., it is defined coinductively by

$$\begin{aligned} e \sim_X t \overset{\text{def}}{\iff}\ & \forall \sigma : X | Var \underset{\text{fin}}{\rightharpoonup} Arg.\ \text{let } \ell = \max(\operatorname{lv} e, \operatorname{lv} t) \text{ in} \\ & (\sigma e \Downarrow^\ell \Longleftrightarrow \sigma t \Downarrow^\ell) \wedge (\sigma e \Downarrow^\ell u \wedge \sigma t \Downarrow^\ell v \Longrightarrow u\ \{\overline{(\sim_X)}\}^\ell\ v) \end{aligned}$$

which is just Equation (A 2) with the first $\Longrightarrow$ replaced by $\Longleftrightarrow$ and R replaced by $\sim$.

The following proposition shows that indexed applicative bisimilarity agrees with the simpler notion of indexed applicative mutual similarity, which is the symmetric reduction of $\overline{(\lesssim_X)}$. We will use these notions interchangeably.

Proposition 57. If $\overline{(\sim'_X)} \overset{\text{def}}{=} \overline{(\lesssim_X)} \cap \overline{(\gtrsim_X)}$, then $\overline{(\sim_X)} = \overline{(\sim'_X)}$.

Proof. Follows straightforwardly from the fact that small-step evaluation is deterministic in λ^U. □

As discussed above, the main idea is that indexed applicative bisimilarity should be a re-definition of observational equivalence. However, indexed applicative bisimilarity coincides not with observational equivalence but an indexed variant thereof. At each index X, the relation $(\lessapprox_X)$ asserts $(\lessapprox)$ under substitutions whose domains contain X. Then, whereas Howe proved $(\lesssim) \subseteq (\lessapprox)$, we prove $\overline{(\lesssim_X)} \subseteq \overline{(\lessapprox_X)}$.

Definition 58 (Indexed observational order and equivalence)**.** Define $e \approx_X t \overset{\text{def}}{\iff} \forall \sigma : X | Var \underset{\text{fin}}{\rightharpoonup} Arg.\ \sigma e \approx \sigma t$ and $e \lessapprox_X t \overset{\text{def}}{\iff} \forall \sigma : X | Var \underset{\text{fin}}{\rightharpoonup} Arg.\ \sigma e \lessapprox \sigma t$.

Definition 59 (Term constructor)**.** Let τ range over multi-contexts (contexts with zero or more distinguishable holes) of the forms x, c, $(\lambda x.\bullet_1)$, $(\bullet_1\ \bullet_2)$, $\langle\bullet_1\rangle$, $\tilde{}\bullet_1$, and $!\bullet_1$. The two holes in $\bullet_1\ \bullet_2$ can be plugged by different expressions.

Using the notation for sequences and families, we write $\tau\overline{e_i}^{i\in I}$ with $I \in \{\emptyset, \{1\}, \{1,2\}\}$ for a term that is formed by plugging the holes of τ with immediate subterms $\overline{e_i}$. For example, if $\tau \equiv \bullet_1\ \bullet_2$, then $\tau\overline{e_i}^{i\in\{1,2\}} \equiv e_1\ e_2$.

To prove $\overline{(\lesssim_X)} = \overline{(\widehat{\lesssim}_X)}$, we show mutual containment. The harder direction $\overline{(\lesssim_X)} \subseteq \overline{(\widehat{\lesssim}_X)}$, i.e., soundness of indexed bisimilarity, derives from the fact that $\overline{(\lesssim_X)}$ is context-respecting, in the following adapted sense.

Definition 60. An indexed family of relations $\overline{R_X}$ *respects contexts with diminishing indices* iff we have $\forall i.\ e_i R_X t_i \Longrightarrow (\tau\ \overline{e_i}) R_Y (\tau\ \overline{t_i})$, where $Y = X \setminus \{x\}$ if $\tau\ \overline{e_i} \equiv \lambda x.e^0$ and $Y = X$ otherwise.

Theorem 61 (Soundness of indexed applicative bisimulation)**.** $\overline{(\lesssim_X)} \subseteq \overline{(\widehat{\lesssim}_X)}$, therefore $\overline{(\sim_X)} \subseteq \overline{(\approx_X)}$.

Proof. We will show below that $\overline{(\widehat{\lesssim}_X)}$ respects contexts with diminishing indices. Suppose $e \lesssim_X t$. Then, for any $\sigma : X | Var \xrightarrow[\text{fin}]{} $ and any C such that $C[\sigma e], C[\sigma t] \in Prog$, we have

$\sigma e \lesssim_\emptyset \sigma t$	by inspection of Definition 56
$C[\sigma e] \lesssim_\emptyset C[\sigma t]$	by context-respecting property
$C[\sigma e] \Downarrow^0 \Longrightarrow C[\sigma t] \Downarrow^0$ and $\forall c.\ C[\sigma e] \Downarrow^0 c \Longrightarrow C[\sigma t] \Downarrow^0 c$	because $(\lesssim_X) = [\lesssim]_X$

so

$\sigma e \mathrel{\widehat{\lesssim}} \sigma t$	because C was arbitrary
$e \mathrel{\widehat{\lesssim}_X} t$	because σ was arbitrary

Therefore, $(\lesssim_X) \subseteq (\widehat{\lesssim}_X)$ and $(\sim_X) = (\gtrsim_X) \cap (\lesssim_X) \subseteq (\widehat{\gtrsim}_X) \cap (\widehat{\lesssim}_X) = (\approx_X)$. □

To prove that $(\lesssim_X)$ respects contexts with diminishing indices, Howe's precongruence candidate is modified for indexed relations as follows.

Definition 62 (Indexed precongruence candidate)**.** Given a family of relations $\overline{R_X}$, define the *indexed precongruence candidate* $\widehat{R}_X$ by the following rules. The base cases are when τ is of the form x or c.

$$\frac{\tau \not\equiv (\lambda x.\bullet_1) \qquad \forall i.\ e_i\ \widehat{R}_X\ s_i \qquad \tau\overline{s_i}\ R_X\ t}{\tau\overline{e_i}\ \widehat{R}_X\ t} \qquad \frac{e^0\ \widehat{R}_X\ s \qquad (\lambda x.s)\ R_{X\setminus\{x\}}\ t}{(\lambda x.e^0)\ \widehat{R}_{X\setminus\{x\}}\ t}$$

Proposition 63, proposition 64 (iv), and Lemma 65 below imply $\overline{(\widehat{\lesssim}_X)} = \overline{(\lesssim_X)}$, so by proposition 64 (ii), it follows that $\overline{(\lesssim_X)}$ respects contexts with diminishing indices.

Proposition 63. Indexed applicative similarity is a monotonic family of precongruences:

i. $(\lesssim_X)$ is reflexive for every X.
ii. $(\lesssim_X)$ is transitive for every X.
iii. $\overline{(\lesssim_X)}$ is monotonic in X (i.e., $X \subseteq Y \implies (\lesssim_X) \subseteq (\lesssim_Y)$).

Proof. The proofs for i. and ii. are adapted from Howe (1996).

i. Define $(\equiv_X)$ to be syntactic equality for every X. Clearly, $(\equiv_X) \subseteq [\equiv_X]$, so by coinduction $\overline{(\equiv_X)} \subseteq \overline{(\lesssim_X)}$ in the product lattice $\prod_X \wp(E \times E)$. Therefore, $\forall X.\ (\equiv) \subseteq (\lesssim_X)$.
ii. Define $R \circ S \stackrel{\text{def}}{=} \{(e,t) : \exists d.\ eRdSt\}$. Take any triple e, d, t such that $e \lesssim_X d \lesssim_X t$, and let $\sigma : X | Var \underset{\text{fin}}{\rightharpoonup} A, \ell$ be given. Then, $\sigma e \Downarrow^\ell v \implies \sigma d \Downarrow^\ell w \implies \sigma t \Downarrow^\ell u$ and $v\ \{\overline{\lesssim_X}\}^\ell\ w\ \{\overline{\lesssim_X}\}^\ell\ u$. Then, $v\ \{\overline{\lesssim_X \circ \lesssim_X}\}^\ell\ u$, which gives $e\ [\overline{\lesssim_X \circ \lesssim_X}]\ t$. Then, by coinduction $\overline{(\lesssim_X \circ \lesssim_X)} \subseteq \overline{(\lesssim_X)}$.
iii. Suppose $e \lesssim_X t$ and $X \subseteq Y$. Any $\sigma : Y | Var \underset{\text{fin}}{\rightharpoonup} A$ also satisfies $\sigma : X | Var \underset{\text{fin}}{\rightharpoonup} A$, so if $\sigma e, \sigma t \in E^\ell$, then $\sigma e \Downarrow^\ell v \implies \sigma t \Downarrow^\ell u$ where $v\ \{\overline{\lesssim_Z}^Z\}^\ell\ u$. Thus, $e \lesssim_Y t$. □

Proposition 64 (Basic properties of indexed precongruence candidate)**.** 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. $\widehat{R}_X$ is reflexive for every X.
ii. $\overline{\widehat{R}_X}$ respects contexts with diminishing indices.
iii. $e\ \widehat{R}_X\ s\ R_X\ t \implies e\ \widehat{R}_X\ t$ at each X.
iv. $\overline{R_X} \subseteq \overline{\widehat{R}_X}$.

Proof.

i. Induction on e shows $e\ \widehat{R}_X\ e$.
ii. By reflexivity of R_X, derivation rules for $\overline{\widehat{R}_X}$ subsume this assertion.
iii. Straightforward induction on e using i. and transitivity of R_X.
iv. Apply i. to iii. □

Lemma 65. $e\ \widehat{\lesssim}_X\ t \implies e\ [\widehat{\lesssim}_X]\ t$.

Proof. Fix a σ and an ℓ, and assume $\sigma e \overset{\ell}{\rightsquigarrow}{}^* v$ where σe takes n small-steps. Then, we can show $\sigma t \Downarrow^\ell u \wedge v\ \{\overline{\widehat{\lesssim}_X}\}^\ell\ u$ by lexicographic induction on (n, e) with case analysis on the form of e. □

This concludes the soundness proof of applicative bisimilarity. To prove completeness, i.e., $\overline{(\lesssim_X)} \subseteq \overline{(\lesssim_X)}$, we show $\overline{(\lesssim_X)} \subseteq \overline{[\lesssim]_X}$ and coinduct. While proving $\overline{(\lesssim_X)} \subseteq \overline{[\lesssim]_X}$, it is necessary to convert $(\lesssim)$ to $(\lesssim_\varnothing)$. For example, when $e \lesssim_X t$, $\sigma e \Downarrow^0 \langle e' \rangle$ then we can easily show $\sigma t \Downarrow^0 \langle t' \rangle$ with $e' \lesssim t'$, but we cannot immediately conclude the $e' \lesssim_\varnothing t'$ that we need for $\langle e' \rangle\ \{\overline{\lesssim_X}\}^0\ \langle t' \rangle$. The argument given in Proposition 11, which hinges on the new, generalized E_U rule, allows us to perform this conversion from $(\lesssim)$ to $(\lesssim_\varnothing)$. We restate Proposition 11 here for $(\lesssim)$.

Lemma 66. $\forall \sigma : Var \underset{\text{fin}}{\rightharpoonup} Arg.\ e \lesssim t \implies \sigma e \lesssim \sigma t$.

Proof. Use the same argument as Proposition 11. □

Lemma 67. For every X, $(\lessapprox) \subseteq (\lessapprox_X)$. In particular, $(\lessapprox) = (\lessapprox_\emptyset)$. Likewise, $(\approx) \subseteq (\approx_X)$ and $(\approx) = (\approx_\emptyset)$.

Proof. If $e \lessapprox t$, then $\sigma e \lessapprox \sigma t$ for every $\sigma : X|Var \xrightarrow[\text{fin}]{} Arg$ by Lemma 66, so $e \lessapprox_X t$. Therefore, $(\lessapprox) \subseteq (\lessapprox_X)$. When $X = \emptyset$, the reverse containment $(\lessapprox_\emptyset) \subseteq (\lessapprox)$ also holds: The $(\lessapprox_\emptyset)$ relation implies $(\lessapprox)$ under any substitution, including the empty substitution. Hence, we have $(\lessapprox) = (\lessapprox_\emptyset)$. The statement for $(\approx)$ follows immediately. □

Theorem 68 (Completeness of indexed applicative bisimulation)**.** $\overline{(\lessapprox_X)} = \overline{(\lesssim_X)}$ and $\overline{(\approx_X)} = \overline{(\sim_X)}$.

Proof. By Theorem 61, only $\overline{(\lessapprox_X)} \subseteq \overline{(\lesssim_X)}$ and $\overline{(\approx_X)} \subseteq \overline{(\sim_X)}$ need to be proved. Suppose $e \lessapprox_X t$ and fix a $\sigma : X|Var \xrightarrow[\text{fin}]{} Arg$ and an ℓ. By definition, $\sigma e \lessapprox \sigma t$ so $\sigma e \Downarrow^\ell v \Longrightarrow \sigma t \Downarrow^\ell u$; we will show that if these v, u exist then $v \ \{\overline{\lessapprox_X}\}^\ell \ u$.

- ▷ [If $\ell > 0$] Because $v \approx \sigma e \lessapprox \sigma t \approx u$, by Lemma 67 it follows that $v \lessapprox_\emptyset u$.
- ▷ [If $\ell = 0$] Split cases by the form of v.
  - ▷ [If $v \equiv \lambda x.e'$] If u were of the form $\langle d \rangle$, then the context $\langle \tilde{} \bullet \rangle$ would distinguish v and u because $\langle \tilde{} \lambda x.e' \rangle$ is stuck while $\langle \tilde{} \langle d \rangle \rangle \Downarrow^0 \langle d \rangle$. If u were a constant, then the trivial context $\bullet$ would distinguish u and v. Therefore, $u \equiv \lambda x.t'$ for some $t' \in E^0$. For any $a \in Arg$, the relation $v \lessapprox u$ implies $[a/x]e' \lessapprox v \ a \lessapprox u \ a \lessapprox [a/x]t'$ so using Lemma 67, $e' \lessapprox_{\{x\}} t'$.
  - ▷ [If $v \equiv \langle e' \rangle$] By the same argument as above, $u \equiv \langle t' \rangle$. Then, since $e' \in E^0$, we have $e' \approx ! \langle e' \rangle \lessapprox ! \langle t' \rangle \approx t'$, so by Lemma 67, $e' \lessapprox_\emptyset t'$.
  - ▷ [If $v \equiv c$] $u \equiv c$, for otherwise the trivial context $\bullet$ would distinguish u and v.

It follows that $e \, [\lessapprox]_X t$, so $\overline{(\lessapprox_X)} \subseteq \overline{[\lessapprox]_X}$. By coinduction, $\overline{(\lessapprox_X)} \subseteq \overline{(\lesssim_X)}$. Therefore, we have $(\approx_X) = (\gtrsim_X) \cap (\lessapprox_X) = (\gtrsim_X) \cap (\lesssim_X) = (\sim_X)$ for each X. □

Finally, from Theorems 61 and 68, we can prove Theorem 35, the soundness and completeness of non-indexed applicative bisimulations.

Proof of Theorem 35. For the reader's convenience, let us recall the theorem's statement:

> For a substitution-closed binary relation $R \subseteq E \times E$, we have $R \subseteq (\approx)$ iff R is contained in an applicative bisimulation.

Let us first prove soundness: If a substitution-closed R is a non-indexed bisimulation, then $R \subseteq (\approx)$. Given a relation R, define an indexed family $\overline{R_X}$ by $e \ R_X \ t \stackrel{\text{def}}{\Longleftrightarrow} \forall \sigma : X|Var \xrightarrow[\text{fin}]{} Arg. \ (\sigma e) \ R \ (\sigma t)$, and set $\forall X. \ \widetilde{R}_X \stackrel{\text{def}}{=} R_X \cup (\approx_X)$. Note $R = R_\emptyset$. Definition 34 states that R is a (non-indexed) applicative bisimulation precisely when $R \subseteq [\widetilde{R}]_\emptyset \cap [\widetilde{R}^{-1}]_\emptyset$. When this containment holds, we claim that $\forall X. \ \widetilde{R}_X \subseteq [\widetilde{R}]_\emptyset \cap [\widetilde{R}^{-1}]_\emptyset$ follows. Suppose $e \ \widetilde{R}_X \ t$ for some e, t, X, let $\ell = \max(\mathrm{lv}\, e, \mathrm{lv}\, t)$, and let $\forall \sigma : X|Var \xrightarrow[\text{fin}]{} Arg$ be given. Then, we have

$$\exists u. \ \sigma e \Downarrow^\ell u \Longleftrightarrow \exists v. \ \sigma t \Downarrow^\ell v \text{ and if } u, v \text{ exist then } u \ \{\overline{\widetilde{R}_X}\}^\ell \ v \wedge v \ \{\overline{\widetilde{R}_X^{-1}}\}^\ell \ u \qquad \text{(A 3)}$$

as follows. First, we have $\sigma e \; \widetilde{R}_\varnothing \; \sigma t$ from $e \; \widetilde{R}_X \; t$, hence either $\sigma e R_\varnothing \sigma t$ or $\sigma e \approx_\varnothing \sigma t$.

- ▷ [If $\sigma e R_\varnothing \sigma t$] $R_\varnothing = R \subseteq [\widetilde{R}]_\varnothing \cap [\widetilde{R}^{-1}]_\varnothing^{-1}$, so claim (A 3) is immediate.
- ▷ [If $\sigma e \approx_\varnothing \sigma t$] By Theorems 61 and 68 $(\approx_\varnothing) = (\sim_\varnothing)$, so $\sigma e \Downarrow^\ell u$ iff $\sigma t \Downarrow^\ell v$ and $u \; \{\overline{\sim_X}\}^\ell \; v$. But $\overline{(\sim_X)} \subseteq \overline{\widetilde{R}_X}$, so claim (A 3) follows.

Therefore, it follows that $\overline{\widetilde{R}_X} \subseteq [\widetilde{R}]_\varnothing \cap [\widetilde{R}^{-1}]_\varnothing^{-1}$. By coinduction, $\overline{\widetilde{R}_X} \subseteq \overline{(\sim_X)} = \overline{(\approx_X)}$, so in particular $R_\varnothing \subseteq (\widetilde{R}_\varnothing) \subseteq (\approx_\varnothing) = (\approx)$ using Lemma 67.

Now, let us prove completeness: If $R \subseteq (\approx)$ is substitution-closed, then R is a non-indexed bisimulation. Let $R \subseteq (\approx)$ be given, and define the families $\overline{R_X}$ and $\overline{\widetilde{R}_X}$ as above. Then, at each X, we have $R_X \subseteq (\approx_X)$, so $\widetilde{R}_X = (\approx_X) = (\sim_X)$. Therefore, $R = R_\varnothing \subseteq (\sim_\varnothing) \subseteq [\widetilde{R}]_\varnothing \cap [\widetilde{R}^{-1}]_\varnothing^{-1}$, which means that R is a bisimulation. □

A.3 Proofs for inlining fold

This section gives more details for the correctness proofs in Section 6.2. These proofs are not found in the thesis (Inoue, 2012), which predates the material presented in Section 6.2.

A.3.1 Proof of CBN correctness

Proof of Proposition 45. We recall the statement of Proposition 45 here for the reader's convenience:

> In CBV, let $f \in V^0$ and $D \subseteq \mathit{Const}$, and let $D^* \stackrel{\text{def}}{=} \{[d_1, d_2, \ldots, d_n] \mid d_1, \ldots, d_n \in D, \; n \in \mathbb{N}\}$. Assume that $\forall x, y \notin \mathrm{FV}(f). \; \exists e. \; \lambda_{\mathbf{v}}^U \vdash f \; \langle y \rangle \; \langle x \rangle = \langle\!\| e \|\!\rangle$ and, for the same e,
>
> $$\forall c, d \in D. \; \lambda_{\mathbf{v}}^U \vdash \| f \| \; c \; d \Downarrow^0 D \wedge [c, d/y, x] \| e \| \Downarrow^0 D. \qquad (20)$$
>
> Then, for any $d \in D$ and $l \in D^*$, we have $\lambda_{\mathbf{v}}^U \vdash \texttt{stfold} \; f \; d \; l = \texttt{fold} \; \| f \| \; d \; l$.

Starting with `stfold` f, we can β-reduce the application, reduce away the escape with E_U and the assumption $f \; \langle y \rangle \; \langle x \rangle = \langle\!\| e \|\!\rangle$, and then eliminate the `!` by R_U, which gives

```
stfold f =
          let rec loop y xs = match xs with
            | [] -> y
            | x::xs -> loop ‖e‖ xs
          in loop
```

whose right-hand side is unstaged, so by the Erasure Theorem, specifically Lemma 21, it follows that `stfold` $f = \|$`stfold` $f \|$. We then claim $\|$`stfold` $f \| \approx$ `fold` $\| f \|$, which requires an explicit proof because these terms are not identical: $\|$`stfold` $f \|$ passes around two parameters in the loop whereas `fold` $\| f \|$ passes around three. By Proposition 36 (extensionality), it suffices to prove $\forall a, l \in E^0$. $\|$`stfold` $f \| \; a \; l =$ `fold` $\| f \| \; a \; l$. By induction on the number of steps till termination, one can

show that both sides of the equation terminate iff $l \Downarrow^0 [c_1, c_2, \ldots, c_n]$ for some $c_i \in \mathit{Const}$ and that in that case both $\texttt{fold}\ \|f\|\ a\ l$ and $\|\texttt{stfold}\ f\|\ a\ l$ reduce to $\|f\|\ (\ldots(\|f\|\ (\|f\|\ a\ c_1)\ c_2)\ldots)\ c_n$. □

A.3.2 Proof of CBV correctness using normalization

Proof of Proposition 47. We recall the statement of Proposition 47 here for the reader's convenience:

> In CBV, let $f \in V^0$ and $D \subseteq \mathit{Const}$, and let $D^* \stackrel{\text{def}}{=} \{[d_1, d_2, \ldots, d_n] \mid d_1, \ldots, d_n \in D,\ n \in \mathbb{N}\}$. Assume that $\forall x, y \notin \mathrm{FV}(f).\ \exists e.\ \lambda^U_{\mathbf{v}} \vdash f\ \langle y \rangle\ \langle x \rangle = \langle \|e\| \rangle$ and, for the same e,
>
> $$\forall c, d \in D.\ \lambda^U_{\mathbf{v}} \vdash \|f\|\ c\ d \Downarrow^0 D \wedge [c, d/y, x]\|e\| \Downarrow^0 D \qquad (20)$$
>
> Then, for any $d \in D$ and $l \in D^*$, we have $\lambda^U_{\mathbf{v}} \vdash \texttt{stfold}\ f\ d\ l = \texttt{fold}\ \|f\|\ d\ l$.

Just as in the CBN case, we have $\|\texttt{stfold}\ f\| \approx \texttt{fold}\ \|f\|$, so we only need to show $\texttt{stfold}\ f\ d\ l \approx \|\texttt{stfold}\ f\|\ d\ l$. By the Erasure Theorem, specifically Lemma 26, it suffices to show that both $\texttt{stfold}\ f\ d\ l$ and $\|\texttt{stfold}\ f\|\ d\ l$ terminate to constants.

By assumption, $l \equiv [c_1, c_2, \ldots, c_n]$ for some sequence $c_i \in D$. Then, by assumption (20), a sequence $d_0, \ldots, d_n \in D$ with $d_0 \equiv d$ is determined by $\forall i.\ \|f\|\ d_{i-1}\ c_i \Downarrow^0 d_i$. Inspecting the source code of $\|\texttt{stfold}\|$, we see that $\|\texttt{stfold}\ f\|\ d\ l \Downarrow^0 d_n$. Likewise, using assumption (20), we inductively obtain a sequence $d'_0, d'_1, \ldots, d'_n$ with $d'_0 \equiv d_0 \equiv d$ such that $\forall i.\ [d'_{i-1}, c_i/y, x]\|e\| \Downarrow^0 d'_i$. By the same rewrites as in the CBN case, we have

```
stfold f =
            let rec loop y xs = match xs with
              | [] -> []
              | x::xs -> loop ‖e‖ xs
            in loop
```

By inspection of the right-hand side, we get $\texttt{stfold}\ f\ d\ l \Downarrow^0 d'_n$. Thus, both $\|\texttt{stfold}\ f\|\ d\ l$ and $\texttt{stfold}\ f\ d\ l$ terminate to constants, as desired. □

A.3.3 Proof of CBV correctness using careful equalities

Proof of Proposition 49. We recall the statement of Proposition 49 here for the reader's convenience.

> Let $f \in V^0$ and $D \subseteq V^0$, and assume that $\forall x, y \notin \mathrm{FV}(f).\ \exists e.\ \forall u, v \in V^0.\ \lambda^U_{\mathbf{v}\Downarrow}/[v, u/y, x] \vdash f\ \langle y \rangle\ \langle x \rangle = \langle \|e\| \rangle$. Then, $\lambda^U_{\mathbf{v}} \vdash \texttt{stfold}\ f = \texttt{fold}\ \|f\|$.

As before, $\|\texttt{stfold}\ f\| \approx \texttt{fold}\ \|f\|$, so we need only show $\texttt{stfold}\ f \approx \|\texttt{stfold}\ f\|$. This goal reduces by extensionality (Proposition 36) and equivalence of divergent terms (Lemma 38) to $\forall v_0, l \in V^0.\ \forall \sigma : \mathit{Var} \xrightarrow[\text{fin}]{} V^0.\ \sigma(\texttt{stfold}\ f\ v\ l) \approx_{\Uparrow} \sigma(\|\texttt{stfold}\ f\|\ v\ l)$, but because $\texttt{stfold}$ is closed and V^0 is closed under substitutions, it suffices to show $\forall v_0, l \in V^0.\ \texttt{stfold}\ f\ v\ l \approx_{\Uparrow} \|\texttt{stfold}\ f\|\ v_0\ l$.

Recall that $\forall u, v.\ \lambda^U_{\mathbf{v}\Downarrow}/[v, u/y, x] \vdash f\ \langle y\rangle\ \langle x\rangle = \langle \|e\| \rangle$ means

$$\lambda^U_{\mathbf{v}} \vdash f\ \langle y\rangle\ \langle x\rangle = \langle \|e\| \rangle \text{ and} \tag{A 4}$$

$$\forall u, v \in V^0.\ \lambda^U_{\mathbf{v}} \vdash f\ v\ u = [v, u/y, x]\|e\| \tag{A 5}$$

From Equation (A 4), we get

```
stfold f =  let rec loop y xs = match xs with
              | [] -> y
              | x::xs -> loop ‖e‖ xs
            in loop
```
(A 6)

If $\|\texttt{stfold}\ f\|\ v_0\ l \Uparrow^0$, then from Equations (A 5) and (A 6) we can show `stfold` $f\ v_0\ l \Uparrow^0$ as desired. If not, we have `stfold` $f\ v_0\ l \Downarrow^0$ and we are left to show `stfold` $f\ v_0\ l \approx \|\texttt{stfold}\ f\|\ v_0\ l$. Given `stfold` $f\ v_0\ l \Downarrow^0$, there exist $u_1, \ldots, u_n, v_1, \ldots, v_n \in V^0$ such that $l \equiv [u_1, \ldots, u_n]$ and $\forall i.\ \|f\|\ v_{i-1}\ u_i \Downarrow^0 v_i$, and $\|\texttt{stfold}\ f\|\ v_0\ l \Downarrow^0 v_n$. (We urge the reader to keep in mind that the `loop` function in Equation (A 6) folds the input list with $\|e\|$, whereas the sequence $v_1, \ldots, v_n$ was obtained by folding with $\|f\|$. The whole point of the following discussion is to equate folding with $\|e\|$ and folding with $\|f\|$.) Let L be the context that provides the `let rec` in (A 6), i.e.,

```
L ≡def  let rec loop y xs = match xs with
          | [] -> y
          | x::xs -> loop ‖e‖ xs
        in •
```

which folds with $\|e\|$. Then,

$$\begin{aligned}
\lambda^U_{\mathbf{v}} \vdash v_n &= L[\texttt{loop}\ v_n\ []] \\
&= L[\texttt{loop}\ (\|f\|\ v_{n-1}\ u_n)\ []] && \text{(by the definition of } v_n\text{)} \\
&= L[\texttt{loop}\ ([v_{n-1}, u_n/y, x]\|e\|)\ []] && \text{(by (A 5))} \\
&= L[\texttt{loop}\ v_{n-1}\ [u_n]] && \text{(by } \beta_{\mathbf{v}} \text{ and the definition of } \texttt{loop}\text{)} \\
&= L[\texttt{loop}\ v_{n-2}\ [u_{n-1}, u_n]] && \text{(by repeating the same manipulations)} \\
&\vdots \\
&= L[\texttt{loop}\ v_0\ [u_1, \ldots, u_n]] \\
&= \texttt{stfold}\ f\ v_0\ l.
\end{aligned}$$

Therefore, $\lambda^U_{\mathbf{v}} \vdash \texttt{stfold}\ f\ v_0\ l = v_n = \|\texttt{stfold}\ f\|\ v_0\ l$. □

Remark. This result can be proved more simply by noting that extensionality gives $\|f\| \approx \lambda y\ x.\|e\|$ from $\forall v, u \in V^0.\ \|f\|\ v\ u = [v, u/y, x]\|e\|$, hence that `stfold` f can be rewritten modulo ($\approx$) into $\|\texttt{stfold}\ f\|$. We avoided this approach because it does not generalize to cases where f has a restricted domain, i.e., we can require $\lambda^U_{\mathbf{v}\Downarrow}/\sigma \vdash f\ \langle y\rangle\ \langle x\rangle = \langle \|e\| \rangle$ for only a certain set of values.