Reasoning about Multi-stage Programs[⋆]

Jun Inoue[2] and Walid Taha[1,2]

[1] Halmstad University, SE-301 18 Halmstad, Sweden
[2] Rice University, Houston TX 77005, USA
ji2@rice.edu, walid.taha@hh.se

Abstract. We settle three basic questions that naturally arise when verifying multi-stage functional programs. Firstly, 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. Secondly, 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 facts—for example, that functions agreeing on all arguments are equivalent? We provide a sound and complete notion of applicative bisimulation, which establishes such facts or, in principle, any valid program equivalence. These results greatly improve our understanding of 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 are staged to generate specialized, efficient code. Generic codes are excellent targets for verification because they are verified only once and used many times, improving modularity of the correctness proof. However, few formal studies have considered verifying generators written with MSP, and MSP research has predominantly focused on applications that confirm performance benefits [5,4,12,8,6] and on type systems [28,17,32,16,29,30].

A key assumption behind the use of MSP is that it enhances performance while preserving the structure of the code, and that it therefore does not interfere much with reasoning [18,4]. The power function is a good example of MSP preserving structure, presented here in MetaOCaml syntax.

```
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>.)>.
```

[⋆] Supported in part by NSF grant CCF-0747431; NSF Award # 0747431, "Multi-stage programming for object-oriented languages"; and NSF Award # 0720857, "Building physically safe embedded systems".

The `power` function subsumes all functions of the form `fun x` $\rightarrow$ `x*x*...*x` but incurs recursive calls each time it is called. Staging annotations can eliminate this overhead by unrolling the recursion in `genpow`. Brackets `.<`e`>.` delay an expression e. An escape `.~`e must occur within brackets and causes e to be evaluated without delay. The e should return a code value `.<`e'`>.`, and e' replaces `.~`e. For example if `n` $= 2$, the `genpow n .<z>.` in `stpow` returns a delayed multiplication `.<z*z>.`. This is an open term, but MetaOCaml allows manipulation of open terms under escapes. Run `.!`e compiles and runs the code generated by e, so `stpow 2` evaluates to the closure `fun z` $\rightarrow$ `z*z`, which has no recursion. These annotations in MetaOCaml are hygienic (i.e., preserve static scoping [9]), but are otherwise like LISP's quasiquote, unquote, and eval [20].

This example is typical of MSP usage, where a staged program `stpow` is meant as a drop-in replacement for the unstaged program `power`. Note that if we are given only `stpow`, we can reconstruct the unstaged program `power` by erasing the staging annotations from `stpow`—we say that `power` is the *erasure* of `stpow`. Given the similarity of these programs, if we are to verify `stpow`, we naturally expect `stpow` $\approx$ `power` to hold for a suitable equivalence ($\approx$) and hope to get away with proving that `power` satisfies whatever specifications it has, in lieu of `stpow`. We expect `power` to be easier to tackle, since it has no staging annotations and should therefore be amenable to conventional reasoning techniques designed for single-stage programs. But three key questions must be addressed before we can apply this strategy confidently:

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 `.<`Ω`>.` $\not\approx \Omega$ 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 [27], but are too weak to prove such properties as extensionality (i.e., functions agreeing on all inputs are equivalent). Extensional facts like this are indispensable for reasoning about functions, like `stpow` and `power`.

This paper settles these questions, focusing on 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 [28,29,30]. It also ensures that our results apply to dynamically typed languages [9]. Hygiene is widely accepted as a safety feature, and it ensures many of the nice theoretical properties of MSP, which makes it easy to reason about programs, and which we exploit in this study. We believe imperative MSP is not yet ready for an investigation like this. Types are essential for having sane operational semantics without scope extrusion [16], 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 does not seem to have matured to the level of the functional theory that we build upon here.

1.1 Contributions

We extend previous work on the call-by-name (CBN) multi-stage λ calculus, λ^U [27], 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 interesting. We illustrate clearly 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 a lot can be achieved despite this drawback.

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 [1,10] 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).

Throughout the paper, we emphasize the general insights about MSP that we can gain from our results. The ability to verify staged programs fall out from general principles, which we will demonstrate using the power function as a running example. A technical report [14] gives proof details and discussions that we cut out due to space limitations. This paper is intelligible by itself, but we note throughout the paper what additional information to expect in the report.

The most substantial additional material in the report is a correctness proof of the longest common subsequence (LCS) algorithm, meant for readers who wish to see how the erasure idea fares on more complex programs than `power`. LCS uses a sophisticated code-generation scheme that requires `let`-insertion coupled with continuation-passing style (CPS) and monadic memoization [26]. These features make an exact description of the generated code hard to pin down; nonetheless, a proof similar to that of `power` can be adapted fairly straightforwardly.

2 The λ^U Calculus: Syntax, Semantics, and Equational Theory

This section presents the multi-stage λ calculus λ^U. This is a simple but expressive calculus that models all possible uses of brackets, escape, and run in

$$\begin{array}{llllll}
\textit{Levels} & \ell, m \in \mathbb{N} & \textit{Variables} & x, y \in \textit{Var} & \textit{Constants} & c, d \in \textit{Const}
\end{array}$$

$$\begin{array}{ll}
\textit{Expressions} & e, t \in E ::= c \mid x \mid \lambda x.e \mid e\ e \mid \langle e \rangle \mid \tilde{}\,e \mid !\,e \\
\textit{Exact Level} & \mathrm{lv} : E \to \mathbb{N} \text{ where} \\
& \mathrm{lv}\, x \overset{\text{def}}{=} 0 \quad \mathrm{lv}\, c \overset{\text{def}}{=} 0 \quad \mathrm{lv}(e_1\ e_2) \overset{\text{def}}{=} \max(\mathrm{lv}\, e_1, \mathrm{lv}\, e_2) \quad \mathrm{lv}(\tilde{}\,e) \overset{\text{def}}{=} \mathrm{lv}\, e + 1 \\
& \mathrm{lv}(\lambda x.e) \overset{\text{def}}{=} \mathrm{lv}\, e \quad \mathrm{lv}\langle e \rangle \overset{\text{def}}{=} \max(\mathrm{lv}\, e - 1, 0) \quad \mathrm{lv}(!\,e) \overset{\text{def}}{=} \mathrm{lv}\, e \\
\textit{Stratification} & e^\ell, t^\ell \in E^\ell \overset{\text{def}}{=} \{e : \mathrm{lv}\, e \le \ell\} \\
\textit{Values} & u^0, v^0 \in V^0 ::= c \mid \lambda x.e^0 \mid \langle e^0 \rangle \\
& u^{\ell+1}, v^{\ell+1} \in V^{\ell+1} ::= e^\ell \\
\textit{Programs} & p \in \textit{Prog} \overset{\text{def}}{=} \{e^0 : \mathrm{FV}(e^0) = \varnothing\} \\
\textit{Contexts} & C \in \textit{Ctx} ::= \bullet \mid \lambda x.C \mid C\ e \mid e\ C \mid \langle C \rangle \mid \tilde{}\,C \mid !\,C
\end{array}$$

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

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 [27] to allow arbitrary constants. The CBN equational theory is more or less as in [27], 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 that apply to both evaluation strategies. Syntactic equality (α equivalence) is written ($\equiv$). The set of free variables in e is written $\mathrm{FV}(e)$. For a set S, we write S_{cl} to mean $\{e \in S : \mathrm{FV}(e) = \varnothing\}$.

2.1 Syntax and Operational Semantics

The syntax of λ^U is shown in Figure 1. A term is delayed when more brackets enclose it than do escapes, and a program must not have an escape in any non-delayed region. We track *levels* to model this behavior. A term's exact level $\mathrm{lv}\, e$ is its nesting depth of escapes minus brackets, and a program is a closed, exactly level-0 term. A level-0 value (i.e., a value in a non-delayed region) is a constant, an abstraction, or a code value with no un-delayed region. At level $\ell > 0$ (i.e., inside ℓ pairs of brackets), a value is any lower-level term. Throughout the article, "the set of terms with exact level at most ℓ", written E^ℓ, is a much more useful concept than "the set of terms with exact level exactly ℓ". When we say "e has level ℓ" we mean $e \in E^\ell$, whereas "e has exact level ℓ" means $\mathrm{lv}\, e = \ell$. A context C is a term with exactly one subterm replaced by a hole $\bullet$, and $C[e]$ is the term obtained by replacing the hole with e, with variable capture. Staging annotations use the same nesting rules as LISP's quasiquote and unquote [9], but we stress that they preserve scoping: e.g., $\langle \lambda x.\tilde{}(\lambda x.\langle x \rangle)\rangle \equiv \langle \lambda x.\tilde{}(\lambda y.\langle y \rangle)\rangle \not\equiv \langle \lambda y.\tilde{}(\lambda x.\langle y \rangle)\rangle$.

A term is unstaged if its annotations are erased in the following sense; it is staged otherwise. The `power` function is the erasure of `stpow` modulo η reduction.

Evaluation Contexts (Productions marked $[\phi]$ apply only if the guard ϕ is true.)

$$\begin{array}{lll}
\text{(CBN) } \mathcal{E}^{\ell,m} \in \mathit{ECtx}_{\mathbf{n}}^{\ell,m} & ::= & \bullet[m=\ell] \mid \lambda x.\mathcal{E}^{\ell,m}[\ell>0] \mid \langle \mathcal{E}^{\ell+1,m}\rangle \mid \tilde{}\,\mathcal{E}^{\ell-1,m}[\ell>0] \\
 & \mid & !\,\mathcal{E}^{\ell,m} \mid \mathcal{E}^{\ell,m}\, e^{\ell} \mid v^{\ell}\, \mathcal{E}^{\ell,m}[\ell>0] \mid c\, \mathcal{E}^{\ell,m}[\ell=0] \\
\text{(CBV) } \mathcal{E}^{\ell,m} \in \mathit{ECtx}_{\mathbf{v}}^{\ell,m} & ::= & \bullet[m=\ell] \mid \lambda x.\mathcal{E}^{\ell,m}[\ell>0] \mid \langle \mathcal{E}^{\ell+1,m}\rangle \mid \tilde{}\,\mathcal{E}^{\ell-1,m}[\ell>0] \\
 & \mid & !\,\mathcal{E}^{\ell,m} \mid \mathcal{E}^{\ell,m}\, e^{\ell} \mid v^{\ell}\, \mathcal{E}^{\ell,m}
\end{array}$$

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

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

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

$$\text{SS-}E\ \frac{}{\tilde{}\langle e^0\rangle \underset{1}{\rightsquigarrow} e^0} \qquad \text{SS-}R\ \frac{}{!\,\langle e^0\rangle \underset{0}{\rightsquigarrow} e^0} \qquad \text{SS-}\textsc{Ctx}\ \frac{e^m \underset{m}{\rightsquigarrow} t^m}{\mathcal{E}^{\ell,m}[e^m] \underset{\ell}{\rightsquigarrow} \mathcal{E}^{\ell,m}[t^m]}$$

Fig. 2. Operational semantics of λ^U, parametrized in an interpretation (partial) map $\delta : \mathit{Const} \times \mathit{Const} \rightharpoonup \{v \in V^0_{\mathrm{cl}} : v \equiv \|v\|\}$

Definition 1 (Erasure). Define the erasure $\|e\|$ by

$$\|x\| \stackrel{\text{def}}{\equiv} x \qquad \|c\| \stackrel{\text{def}}{\equiv} c \qquad \|\lambda x.e\| \stackrel{\text{def}}{\equiv} \lambda x.\|e\| \qquad \|\tilde{}\,e\| \stackrel{\text{def}}{\equiv} \|e\|$$

$$\|e_1\ e_2\| \stackrel{\text{def}}{\equiv} \|e_1\|\ \|e_2\| \qquad \|\langle e\rangle\| \stackrel{\text{def}}{\equiv} \|e\| \qquad \|!\,e\| \stackrel{\text{def}}{\equiv} \|e\|$$

The operational semantics is given in Figure 2; examples are provided below. Square brackets denote guards on grammatical production rules; for instance, $\mathit{ECtx}_{\mathbf{n}}^{\ell,m} ::= \bullet[m = \ell] \mid \ldots$ means $\bullet \in \mathit{ECtx}_{\mathbf{n}}^{\ell,m}$ iff $m = \ell$. An ℓ, m-evaluation context $\mathcal{E}^{\ell,m}$ takes a level-m redex and yields a level-ℓ term. Redex contractions are: β reduction at level 0, δ reduction at level 0, run-bracket elimination at level 0, and escape-bracket elimination at level 1. CBN uses SS-β and CBV uses SS-$\beta_{\mathbf{v}}$. All other rules are common to both evaluation strategies.

Small-steps specify the behavior of deterministic evaluators. Every term decomposes in at most one way as $\mathcal{E}^{\ell,m}[t]$ where t is a level-m redex, and the small-step reduct is unique if it exists. The δ reductions are given by a partial map $\delta : \mathit{Const} \times \mathit{Const} \rightharpoonup \{v \in V^0_{\mathrm{cl}} : v \equiv \|v\|\}$, which is undefined for ill-formed pairs like $\delta(\mathtt{not}, \mathtt{5})$. We assume constant applications do not return staged terms.

The difference between CBV and CBN evaluation contexts is that CBV can place the hole inside the argument of a level-0 application, but CBN 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 it is a constant. At level > 0, both evaluation strategies simply walk over the syntax tree of the delayed term to look for escapes, including ones that occur inside the arguments of applications.

Notation. We write $\lambda^U_{\mathbf{n}} \vdash e \underset{\ell}{\rightsquigarrow} t$ for a CBN small-step judgment and $\lambda^U_{\mathbf{v}} \vdash e \underset{\ell}{\rightsquigarrow} t$ for CBV. We use similar notation for $(\Downarrow)$, $(\Uparrow)$, and $(\approx)$ defined below. For any relation R, let R^* be its reflexive-transitive closure. The metavariables $a, b \in \mathit{Arg}$ will range over substitutable arguments, i.e., e^0 for CBN and v^0 for CBV.

For example, $p_1 \equiv (\lambda y.\langle 40 + y\rangle)\ (1+1)$ is a program. Its value is determined by $(\underset{0}{\rightsquigarrow})$, which works like in conventional calculi. In CBN, $\lambda^U_{\mathbf{n}} \vdash p_1 \underset{0}{\rightsquigarrow} \langle 40 + (1+1)\rangle$. The redex $(1+1)$ is not selected for contraction because $(\lambda y.\langle 40{+}y\rangle)\ \bullet \notin \mathit{ECtx}^{0,0}_{\mathbf{n}}$. In CBV, $(\lambda y.\langle 40 + y\rangle)\ \bullet \in \mathit{ECtx}^{0,0}$, so $(1 + 1)$ is selected for contraction: $\lambda^U_{\mathbf{v}} \vdash p_1 \underset{0}{\rightsquigarrow} (\lambda y.\langle 40 + y\rangle)\ 2 \underset{0}{\rightsquigarrow} \langle 40 + 2\rangle$.

Let $p_2 \equiv \langle \lambda z.z\ (\tilde{}[(\lambda\_.\langle z\rangle)\ 1])\rangle$, where we used square brackets [ ] as parentheses to improve readability. Let e^0 be the subterm inside square brackets. In both CBN and CBV, p_2 decomposes as $\mathcal{E}[e^0]$, where $\mathcal{E} \equiv \langle \lambda z.z\ (\tilde{}\bullet)\rangle \in \mathit{ECtx}^{0,0}$, and e^0 is a level-0 redex. Note the hole of $\mathcal{E}$ is under a binder and the redex e^0 is open, though p_2 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 until the delay is canceled by !. Hence, $p_2 \underset{0}{\rightsquigarrow} \langle \lambda z.z\ (\tilde{}\langle z\rangle)\rangle \underset{0}{\rightsquigarrow} \langle \lambda z.z\ z\rangle$.

As usual, this "untyped" formalism can be seen as dynamically typed. In this view, ˜ 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; see Section 3.

Remark. Binary operations on constants are modeled by including their partially applied variants. To model addition we take $\mathit{Const} \supseteq \mathbb{Z} \cup \{+\} \cup \{+_k : k \in \mathbb{Z}\}$ and set $\delta(+, k) = +_k$, $\delta(+_k, k') =$ (the sum of k and k'). For example, in prefix notation, $(+\ 3\ 5) \underset{0}{\rightsquigarrow} (+_3\ 5) \underset{0}{\rightsquigarrow} 8$. Conditionals are modeled by taking $\mathit{Const} \supseteq \{\texttt{()}, \texttt{true}, \texttt{false}, \texttt{if}\}$ and setting $\delta(\texttt{if}, \texttt{true}) = \lambda a.\lambda b.a\ \texttt{()}$ and $\delta(\texttt{if}, \texttt{false}) = \lambda a.\lambda b.b\ \texttt{()}$. Then, e.g., $\texttt{if true}\ (\lambda\_.1)\ (\lambda\_.0) \underset{0}{\rightsquigarrow} (\lambda a.\lambda b.a\ \texttt{()})\ (\lambda\_.1)\ (\lambda\_.0) \underset{0}{\rightsquigarrow}^* 1$.

Definition 2 (Termination and Divergence). An $e \in E^\ell$ *terminates* to $v \in V^\ell$ at level ℓ iff $e \underset{\ell}{\rightsquigarrow}^* v$, 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.

The operational semantics induces the usual notion of observational equivalence, which relate terms that are interchangeable under all program contexts.

Definition 3 (Observational Equivalence). $e \approx t$ iff for every C such that $C[e], C[t] \in \mathit{Prog}$, $C[e] \Downarrow^0 \Longleftrightarrow C[t] \Downarrow^0$ holds and whenever one of them terminates to a constant, the other also terminates to the same constant.

2.2 Equational Theory

The equational theory of λ^U is a proof system containing four inference rules: compatible extension ($e = t \Longrightarrow C[e] = C[t]$), reflexivity, symmetry, and transitivity. The CBN axioms are $\lambda^U_{\mathbf{n}} \overset{\text{def}}{=} \{\beta, E_U, R_U, \delta\}$, while CBV axioms are

$\lambda^U_{\mathbf{v}} \stackrel{\text{def}}{=} \{\beta_{\mathbf{v}}, E_U, R_U, \delta\}$. Each axiom is shown 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. 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	$c\ d = \delta(c, d)$	$(c, d) \in \operatorname{dom}\delta$

For example, axiom $\beta_{\mathbf{v}}$ gives $\lambda^U_{\mathbf{v}} \vdash (\lambda\_.0)\ 1 = 0$. 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 $\langle(\lambda\_.0)\ 1\rangle \underset{0}{\not\rightsquigarrow} \langle 0\rangle$ because brackets delay the application, but reduction allows all left-to-right rewrites by the axioms, so $\langle(\lambda\_.0)\ 1\rangle \longrightarrow \langle 0\rangle$ nonetheless. Intuitively, $\langle(\lambda\_.0)\ 1\rangle \underset{0}{\not\rightsquigarrow} \langle 0\rangle$ because an evaluator does not perform 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). 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 4 (Church-Rosser Property). $e = e' \iff \exists t.\ e \longrightarrow^* t \longleftarrow^* e'$.

Provable equality is an approximation of observational equivalence. The containment $(=) \subset (\approx)$ is proper because $(\approx)$ is not semi-decidable (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".

Theorem 5 (Soundness). $(=) \subset (\approx)$.

Theorem 6. If $e \in E^\ell, v \in V^\ell$, then $e \Downarrow^\ell v \implies (e \longrightarrow^* v \wedge e = v)$ and $e = v \in V^\ell \implies (\exists u \in V^\ell.u = v \wedge e \longrightarrow^* u \wedge e \Downarrow^\ell u)$.

The CBN version of the equational theory given here is not identical to [27], but generalizes the E_U rule from $\tilde{}\langle e^0\rangle = e^0$ to $\tilde{}\langle e\rangle = e$. This minor generalization comes in handy for eliminating redundant escapes and brackets. An example is found in the proof that substitution preserves $(\approx)$:

Proposition 7. $e \approx t \implies [a/x]e \approx [a/x]t$.

Proof. The idea is to plug e, t into the context $(\lambda x.\bullet)\ a$ and to apply $\beta/\beta_{\mathbf{v}}$ to get $[a/x]e = (\lambda x.e)\ a \approx (\lambda x.t)\ a = [a/x]t$. However, $\beta/\beta_{\mathbf{v}}$ does not apply if e, t are not level 0, so we have to make them level 0. Take $\ell = \max(\mathrm{lv}\, e, \mathrm{lv}\, t)$. Then

$$(\lambda x.\langle\langle \cdots \langle e \rangle \cdots \rangle\rangle)\ a \approx (\lambda x.\langle\langle \cdots \langle t \rangle \cdots \rangle\rangle)\ a\ , \tag{1}$$

where e and t are each enclosed in ℓ pairs of brackets. Now $\beta/\beta_{\mathbf{v}}$ applies, and we get $\langle\langle \cdots \langle [a/x]e \rangle \cdots \rangle\rangle \approx \langle\langle \cdots \langle [a/x]t \rangle \cdots \rangle\rangle$. Escaping both sides ℓ times gives

$$\tilde{}\,\tilde{}\cdots\tilde{}\langle\langle \cdots \langle [a/x]e \rangle \cdots \rangle\rangle \approx \tilde{}\,\tilde{}\cdots\tilde{}\langle\langle \cdots \langle [a/x]t \rangle \cdots \rangle\rangle\ . \tag{2}$$

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 $\langle\langle \cdots \langle [a/x]e \rangle \cdots \rangle\rangle$ part increases—so the generalization is strictly necessary. □

Theorem 7 shows that applying substitutions to an equivalence does not compromise its validity. This fact plays a role in the completeness proof of applicative bisimulation (to be introduced in Section 5), but we will leave those details to the technical report. The more interesting, and surprising, fact is that the converse fails in $\lambda^U_{\mathbf{v}}$—we cannot in general conclude $e \approx t$ from $\forall a.\ [a/x]e \approx [a/x]t$. We will discuss this issue in Section 3.

Remark. R_U and $\beta/\beta_{\mathbf{v}}$ cannot be generalized in a similar fashion as they involve demotion—moving a term from one level to another. If we generalized R_U to $!\,\langle e \rangle = e$, the e on the left appears in more brackets than on the right, so on the left we need more escapes than on the right to un-delay a subterm of e. For instance, if t is some divergent level-0 term, $\langle !\,\langle \tilde{}t \rangle\rangle = \langle \tilde{}t \rangle$ is an instance of the generalized R_U rule, but $\langle !\,\langle \tilde{}t \rangle\rangle \Downarrow^0$ while $\langle \tilde{}t \rangle \Uparrow^0$. The correct R_U rule avoids this problem by restricting e to level 0, thus $!\,\langle e^0 \rangle = e^0$. The technical report proves that equational rules entailing unrestricted demotion are always unsound.

3 Closing Substitutions Compromise Validity

Here is a striking example of how reasoning in λ^U differs from reasoning in single-stage calculi. Traditionally, CBV calculi admit the equational rule

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

Plotkin's seminal λ_V [22], 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 $\lambda^U_{\mathbf{v}}$. For example, the terms $(\lambda\_.0)\ x$ and 0 may seem interchangeable, but in $\lambda^U_{\mathbf{v}}$ they are distinguished by the program context $\mathcal{E} \overset{\mathrm{def}}{\equiv} \langle \lambda x.\tilde{}[(\lambda\_.\langle 1 \rangle)\ \bullet] \rangle$:

$$\langle \lambda x.\tilde{}[(\lambda\_.\langle 1 \rangle)\ ((\lambda\_.0)\ x)] \rangle \Uparrow^0 \ \text{ but } \langle \lambda x.\tilde{}[(\lambda\_.\langle 1 \rangle)\ 0] \rangle \Downarrow^0 \langle \lambda x.1 \rangle\ . \tag{3}$$

(Once again, we are using [ ] as parentheses to enhance readability.) The term on the left is stuck because $x \notin V^0$ and $x \underset{0}{\not\rightarrow}$. Intuitively, the value of x is demanded before anything is substituted for it. 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 $\lambda^U_{\mathbf{v}}$ can evaluate open terms. Some readers 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 $\lambda^U_{\mathbf{v}}$ including MetaOCaml have no runtime values, or data structures, representing the variable x—they implement $x \notin V^0$. They never perform $(\lambda\_.0)\ x \underset{0}{\rightsquigarrow} 0$, 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 error. Some program contexts in λ^U do force the evaluation of open terms, e.g., the $\mathcal{E}$ given above. We must then define a small-step semantics with $(\lambda\_.0)\ x \underset{0}{\not\rightsquigarrow} 0$, or else we would not model actual implementations, and we must reject β_x, for it is unsound for $(\approx)$ in such a small-step semantics. In other words, lack of β_x is an inevitable consequence of the way 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 always no 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 $(\approx)$.

Now, the general, more important, problem is that reasoning under substitutions is unsound, i.e., $\forall\sigma.\ \sigma e \approx \sigma t \not\Longrightarrow e \approx t$. The lack of β_x is just an example of how this problem shows up in reasoning. We stress that the real challenge is this more general problem with substitutions 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. But this little victory over β_x does not justify reasoning under substitutions, and how or whether we can achieve the latter is a much more difficult question. It is unclear if any type systems justify reasoning under substitutions in general, and it is even less clear how 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 with applicative bisimulation (Section 5) that the lack of β_x is not a large drawback after all, as a refined form of β_x can be used instead:

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

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. $C\beta_x$ 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 $e \approx \|e\|$.

4.1 Theorem Statement

The theorem statement differs for CBN and CBV. Let us see CBN first. The intuition behind the theorem is that all that staging annotations do is to 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 the redex can be chosen from anywhere.[1] Thus, erasure commutes with CBN reductions (Figure 3(a)). The same holds for provable equalities.

Theorem 8 (CBN Erasure). If $\lambda^U_{\mathbf{n}} \vdash e \longrightarrow^* t$ then $\lambda^U_{\mathbf{n}} \vdash \|e\| \longrightarrow^* \|t\|$. Also, if $\lambda^U_{\mathbf{n}} \vdash e = t$ then $\lambda^U_{\mathbf{n}} \vdash \|e\| = \|t\|$.

How does this 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 show in Section 4.2, this search boils down to a termination check on the generator.

Lemma 9 (CBN Correctness). $(\exists t.\; \lambda^U_{\mathbf{n}} \vdash e = \|t\|) \Longrightarrow \lambda^U_{\mathbf{n}} \vdash e = \|e\|$.

CBV satisfies a property similar to Theorem 8, but the situation is more subtle. Staging modifies the evaluation strategy in CBV as well, but not all of them can be simulated in the erasure by CBV reductions, for $\beta_{\mathbf{v}}$ reduces only a subset

[1] This 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.

$$\begin{tikzcd}
\lambda^U_{\mathbf{n}} \vdash e \arrow[r, "*"] \arrow[d, "\|-\|"'] & t \arrow[d, "\|-\|"] \\
\lambda^U_{\mathbf{n}} \vdash \|e\| \arrow[r, "*"] & \|t\|
\end{tikzcd}$$

(a) CBN erasure.

$$\begin{tikzcd}
\lambda^U_{\mathbf{v}} \vdash e \arrow[r, "*"] \arrow[d, "\|-\|"'] & t \arrow[d, "\|-\|"] \\
\lambda^U_{\mathbf{n}} \vdash \|e\| \arrow[r, "*"] & \|t\|
\end{tikzcd}$$

(b) CBV erasure.

$$\begin{tikzcd}
\lambda^U_{\mathbf{n}} \\
\top \\
\lambda^U_{\mathbf{v}} \vdash e \arrow[r, equal] \arrow[d, equal] & c \arrow[d, phantom, "\equiv"] \\
\lambda^U_{\mathbf{v}} \vdash \|e\| \arrow[r, equal] & d
\end{tikzcd}$$

(c) CBV correctness lemma.

Fig. 3. Visualizations of the Erasure Theorem and the derived correctness lemma

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 10 (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 $\lambda^U_{\mathbf{n}} \vdash \|e\| = \|t\|$.

This theorem has similar ramifications as 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 11. $(\exists t.\ \lambda^U_{\mathbf{v}} \vdash e = \|t\|) \Longrightarrow \lambda^U_{\mathbf{n}} \vdash e = \|e\|$.

CBN equalities given by this corollary may at first seem irrelevant to CBV programs, but in fact if we show that e and $\|e\|$ CBV-reduce to constants, then the CBN equality can be safely cast to CBV equality. Figure 3(c) summarizes this reasoning. Given e, suppose we found some c, d that satisfy the two horizontal CBV equalities. Then from the top equality, Theorem 11 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. Tracing the diagram clockwise from the top left corner gives $\lambda^U_{\mathbf{v}} \vdash e = \|e\|$.

Lemma 12 (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.

4.2 Example: Erasing Staged Power

Let us show how the Erasure Theorem applies to `stpow`. First, some technicalities: 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. We assume λ^U has integers and booleans. 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, not the erasure of a variable named `stpow`.

As a caveat, we might want to prove `stpow` $\approx$ `power` but this goal is not quite right. The whole point of `stpow` is to process the first argument without waiting for the second, so it can disagree with `power` when partially applied, e.g., `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 required for Theorem 9. 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 syntactically equal, the former containing an η redex.

Lemma 13. $\lambda^U_{\mathbf{n}} \vdash \|$`stpow`$\| =$ `power`

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

Proposition 14 (Erasing CBN Power). $\forall k \in \mathbb{Z}^+.\ \lambda^U_{\mathbf{n}} \vdash$ `stpow` $k =$ `power` k.

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

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

hence `stpow` $k = \|$`stpow`$\|\ k =$ `power` k by Lemmas 9 and 13. □

The proof for CBV is similar, but we need to fully apply both `stpow` and its erasure to confirm that they both reach some constant. The beauty of Theorem 12 is that we do not have to know what those constants are. Just as in CBN, the erasure $\|$`stpow`$\|$ is equivalent to `power`, but note this part of the proof uses $C\beta_x$.

Lemma 15. $\lambda^U_{\mathbf{n}} \vdash \|$`stpow`$\| \approx$ `power`

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

Proposition 16 (Erasing CBV Power). For $k \in \mathbb{Z}^+$ and $m \in \mathbb{Z}$, $\lambda^U_{\mathbf{v}} \vdash$ `stpow` $k\ m \approx$ `power` $k\ m$.

Proof. We stress that this proof works entirely with CBV equalities; we have no need to deal with CBN once Theorem 12 is established. By induction on k, we prove that $\exists e.$ `genpow` k `.<x>.` $=$ `.<`$\|e\|$`>.` and $[m/\mathtt{x}]\|e\| \Downarrow^0 m'$ 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, so this property is preserved. Then

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

Clearly `power` $k\ m$ terminates to a constant. By Theorem 15, $\|$`stpow`$\|\ k\ m$ also yields a constant, so by Theorem 12, `stpow` $k\ m = \|$`stpow`$\|\ k\ m \approx$ `power` $k\ m$. □

These proofs illustrate our answer to the erasure question in the introduction. Erasure is semantics-preserving if the generator terminates to $\langle\|e\|\rangle$ in CBN, or 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 essentially a modification of the termination proof.

4.3 Why CBN Facts Are Necessary for CBV Reasoning

So far, 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. A natural, alternative idea is to find a subset of CBV reductions that erase to CBV reductions. This alternative approach does work [31,14], but we show here that it only works in simple cases.

The problem with erasing CBV reductions is that the argument in a $\beta_{\mathbf{v}}$ redex may have a divergent erasure. If we restrict $\beta_{\mathbf{v}}$ to

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

which checks that the argument's erasure terminates, then reductions under the axiom set $\lambda^U_{\mathbf{v}\Downarrow} \overset{\text{def}}{=} \{\beta_{\mathbf{v}\Downarrow}, E_U, R_U, \delta\}$ erase to CBV reductions. But $\beta_{\mathbf{v}\Downarrow}$ is much too crude, for it cannot reduce $(\lambda y.e^0)\ \langle x\rangle$ (note $x \Uparrow^0$) and fails to handle programs as simple as `stpow`. A natural solution is to check carefulness under substitutions:

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

Ignoring some technical details, if we let $\lambda^U_{\mathbf{v}\Downarrow}/\sigma \overset{\text{def}}{=} \{\beta_{\mathbf{v}\Downarrow}/\sigma, E_U, R_U, \delta\}$ for any substitution $\sigma : \mathit{Var} \underset{\text{fin}}{\rightharpoonup} V^0$, then $\lambda^U_{\mathbf{v}\Downarrow}/\sigma \vdash e = t$ implies $\lambda^U_{\mathbf{v}} \vdash \sigma e = \sigma t$. This observation suffices to verify `stpow` (see the technical report for a demonstration).

However, careful reductions quickly become unwieldy in the face of binders. For instance, if we write `let` x `=` e `in` t as a shorthand for $(\lambda x.t)\ e$, clearly

`.!.<let` y `= 0 in let` x `=` y `in .~((`$\lambda z.z$`) .<`x`+`y`>.)>.`

is equivalent to its erasure. To prove this, we might observe that $\beta_{\mathbf{v}\Downarrow}/[0, 0/x, y] \vdash (\lambda z.z)$ `.<`x`+`y`>.` $=$ `.<`x`+`y`>.`; however, the "compatible extension",

$$\lambda^U_{\mathbf{v}\Downarrow}/[0, 0/x, y] \vdash \texttt{let } x \texttt{ = } y \texttt{ in .~((}\lambda z.z\texttt{) .<}x\texttt{+}y\texttt{>.)} = \ldots\ ,$$

does not hold because the x in $\lambda^U_{\mathbf{v}\Downarrow}/[0, 0/x, y]$ cannot refer to the x bound in the object term (else we would have to give up hygiene).

In general, we must reason under different substitutions in different scopes, and it is tricky to propagate the results obtained under $\lambda^U_{\mathbf{v}\Downarrow}/\sigma$ to an outer context where some variables in σ may have gone out of scope. While it may not be possible to pull off the bookkeeping, we find ourselves fighting against hygiene rather than exploiting it. In this sense, restricting CBV reductions gives a less useful approach than appealing to CBN reasoning results, especially for programs that generate nested binders. The longest common subsequence found in the technical report is an example of such a generator.

5 Applicative Bisimulation

This section presents applicative bisimulation [1,10], a well-established tool for analyzing higher-order functional programs. Bisimulation is sound and complete for $(\approx)$, and justifies $C\beta_x$ (Section 3) and extensionality, allowing us to handle the divergence issues ignored in Section 4.2.

5.1 Proof by Bisimulation

Intuitively, for a pair of terms to applicatively bisimulate, they must both terminate or both diverge, and if they terminate, their values must bisimulate 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 transition system consisting of evaluation $(\Downarrow)$ and experiments. If eRt implies that either $e \approx t$ or e,t bisimulate, then $R \subseteq (\approx)$.

Definition 17 (Relation Under Experiment). Given a relation $R \subseteq E \times E$, let $\widetilde{R} \overset{\text{def}}{=} R \cup (\approx)$. For $\ell > 0$ set $u\ R^{\ell}_{\dagger}\ v$ iff $u\widetilde{R}v$. For $\ell = 0$ set $u\ R^{0}_{\dagger}\ v$ iff either:

- $u \equiv v \in \mathit{Const}$,
- $u \equiv \lambda x.e$ and $v \equiv \lambda x.t$ for some e, t s.t. $\forall a.([a/x]e)\widetilde{R}([a/x]t)$, or
- $u \equiv \langle e \rangle$ and $v \equiv \langle t \rangle$ for some e, t s.t. $e\widetilde{R}t$.

Definition 18 (Applicative Bisimulation). An $R \subseteq E \times E$ is an *applicative bisimulation* iff every pair $(e, t) \in R$ satisfies the following: let $\ell = \max(\mathrm{lv}\, e, \mathrm{lv}\, t)$; then for any finite substitution $\sigma : \mathit{Var} \underset{\text{fin}}{\rightharpoonup} \mathit{Arg}$ we have $\sigma e \Downarrow^{\ell} \Longleftrightarrow \sigma t \Downarrow^{\ell}$, and if $\sigma e \Downarrow^{\ell} u \wedge \sigma t \Downarrow^{\ell} v$ then $u\ R^{\ell}_{\dagger}\ v$.

Theorem 19. Given $R \subset E \times E$, define $R^{\bullet} \overset{\text{def}}{=} \{(\sigma e, \sigma t) : eRt, (\sigma : \mathit{Var} \underset{\text{fin}}{\rightharpoonup} \mathit{Arg})\}$. Then $R \subseteq (\approx)$ iff $R^{\bullet}$ is an applicative bisimulation.

This is our answer to the extensional reasoning question in the introduction: this theorem shows that bisimulation can in principle derive *all* valid equivalences, including all extensional facts. Unlike in single-stage languages [1,13,10], σ ranges over non-closing substitutions, which may not substitute for all variables or may substitute open terms. Closing substitutions are unsafe since 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 is still a more approachable alternative to $(\approx)$.

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

Proposition 20. 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 $R \overset{\text{def}}{=} \{(\lambda x.e, \lambda x.t)\}^{\bullet}$. 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 Barendregt's variable convention [2], x is fresh for σ, thus $\sigma\lambda x.e \equiv \lambda x.\sigma e$ and $\sigma\lambda x.t \equiv \lambda x.\sigma t$. We must check $[a/x]\sigma e \approx [a/x]\sigma t$: by assumption $\sigma[a/x]e \approx \sigma[a/x]t$, and one can show that σ and $[a/x]$ commute modulo $(\approx)$. Hence by Theorem 19, $\lambda x.e \approx \lambda x.t$. $\square$

Corollary 21 (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 Theorem 20 with $\beta/\beta_{\mathbf{v}}$. $\square$

The proof of Theorem 20 would have failed in CBV had we 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 $\lambda x.t$ do not bisimulate 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.

Howe's method [13] is used to prove Theorem 19, but adapting this method to λ^U is surprisingly tricky because λ^U's bisimulation must handle substitutions inconsistently: in Theorem 18 we cannot restrict our attention to σ's that substitute away any particular variable, but in Theorem 17, for $\lambda x.e\ R^0_{\dagger}\ \lambda x.t$, we must restrict our attention to the case where substitution eliminates x. Proving Theorem 19 entails coinduction on a self-referential definition of bisimulation; however, Theorem 17 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. 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 Theorem 17 can refer to a different member of the family. Theorem 19 is then proved by mutual coinduction. See the technical report for more details.

Remark. Extensionality is a common addition to the equational theory for the plain λ calculus, usually called the ω rule [21,15]. But unlike ω in the plain λ calculus, λ^U functions must agree on open-term arguments as well. This is no surprise since 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.

Remark. The only difference between Theorem 18 and applicative bisimulation in the plain λ calculus is that Theorem 18 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 an erased program. We do not have a proof of this conjecture, however. Conservativity is usually proved through a denotational semantics, which is notoriously difficult to devise for hygienic MSP. It will at least deserve separate treatment from this 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` terminate on 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` $n\ x$.

Lemma 22. Let $e' \approx_{\Uparrow} t'$ mean $e' \approx t' \vee (\sigma e' \Uparrow^{\ell} \wedge \sigma t' \Uparrow^{\ell})$ where $\ell = \max(\mathrm{lv}\, e', \mathrm{lv}\, t')$. For a fixed e, t, if for every $\sigma : Var \underset{\text{fin}}{\rightharpoonup} Arg$ we have $\sigma e \approx_{\Uparrow} \sigma t$, then $e \approx t$.

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

Proposition 23 (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 $\forall e, t \in E^0.\ \forall \sigma : Var \underset{\text{fin}}{\rightharpoonup} Arg.\ \sigma($`power` $e\ t) \approx_{\Uparrow} \sigma($`stpow` $e\ t)$, whence `power` $\approx \lambda n.\lambda x.$`stpow` $n\ x$ by Theorem 22 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 Theorem 14, `power` $e =$ `power` $m \approx$ `stpow` $m =$ `stpow` e, so `power` $e\ t \approx$ `stpow` $e\ t$. □

Proposition 24 (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 Theorem 16 applies. □

Remark. Real code should not use $\lambda n.\lambda x.$`stpow` $n\ x$, as it re-generates 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 Related Works

Taha [27] first discovered λ^U, which showed that functional hygienic MSP admits intensional equalities like β, even under brackets. However, [27] showed the mere existence of the theory and did not explore how to use it for verification, or how to prove extensional equivalences. Moreover, though [27] laid down the operational semantics of both CBV and CBN, it gave an equational theory for only CBN and left the trickier CBV unaddressed.

Yang pioneered the use of an "annotation erasure theorem", which stated $e \Downarrow^0 \langle\|t\|\rangle \implies \|t\| \approx \|e\|$ [31]. But there was a catch: the assertion $\|t\| \approx \|e\|$ 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 and Tratt [3] 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 considered concern relatively simplistic programs. Mini-ML$_e^\Box$ also prohibits manipulating open terms, so it does not capture the challenges of reasoning about free variables, which was one of the main challenges to which we faced up. 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.

For MSP with variable capture, Choi et al. [7] recently 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. 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.

7 Conclusion and Future Work

We addressed three basic concerns for verifying staged programs. We showed that staging is a non-conservative extension because reasoning under substitutions is unsound in a MSP language, even if we are dealing with unstaged terms. Despite this drawback, untyped functional MSP has a rich set of useful properties.

We proved that simple termination conditions guarantee that erasure preserves semantics, which reduces the task of proving the irrelevance of annotations on a program's semantics to the better studied problem of proving termination. We showed a sound and complete notion of applicative bisimulation for this setting, which allows us to reason under substitution in some cases. In particular, the shocking lack of β_x in $\lambda_{\mathbf{v}}^{U}$ is of limited practical relevance as we have $C\beta_x$ instead.

These results improve our general understanding of hygienic MSP. We better know the multi-stage λ calculus' similarities with the plain λ calculus (e.g., completeness of bisimulation), as well as its pathologies and the extent to which they are a problem. The Erasure Theorem gives intuitions on what staging annotations can or cannot do, with which we may educate the novice multi-stage programmer. This understanding has brought us to a level where the proof of a sophisticated generator like LCS is easily within reach.

This work may be extended in several interesting directions. We have specifically identified some open questions about λ^{U}: which type systems allow reasoning under substitutions, whether it is conservative over the plain λ calculus for closed terms, and whether the extensionality principle can be strengthened to require equivalence for only closed-term arguments.

Devising a mechanized program logic would also be an excellent goal. Berger and Tratt's system [3] 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 [11].

Bisimulation is known to work for single-stage imperative languages, though in quite different flavors from applicative bisimulation [19]. Adapting them to MSP would make the emerging imperative hygienic MSP languages [16,24,30] susceptible to analysis. 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 one for tracking mutual (in)dependence of effects, perhaps separation logic [23] for the latter. In any case, investigation of imperative hygienic MSP may have to wait until the foundation matures, as noted in the introduction.

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 [25].

Acknowledgments. We would like to thank Gregory Malecha, Edwin Westbrook, and Mathias Ricken for their input on the technical content of this paper. We are grateful to Veronica Gaspez and Bertil Svensson for their feedback. We thank Halmstad University for their support in completing this work. The first author has received significant guidance and supervision from Robert

Cartwright, Vivek Sarkar, and Marcia O'Malley, who, together with the second author, constitute the first author's doctoral thesis committee. We thank Carol Sonenklar and Ray Hardesty for their input on the writing.

References

1. Abramsky, S.: The Lazy Lambda Calculus, pp. 65–116. Addison-Wesley, Boston (1990)
2. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. Studies in Logic and The Foundations of Mathematics. North-Holland (1984)
3. Berger, M., Tratt, L.: Program Logics for Homogeneous Meta-programming. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 64–81. Springer, Heidelberg (2010)
4. Brady, E., Hammond, K.: A verified staged interpreter is a verified compiler. In: 5th International Conference on Generative Programming and Component Engineering, pp. 111–120. ACM (2006)
5. Carette, J., Kiselyov, O.: Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 256–274. Springer, Heidelberg (2005)
6. Carette, J., Kiselyov, O., Shan, C.-C.: Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 222–238. Springer, Heidelberg (2007)
7. Choi, W., Aktemur, B., Yi, K., Tatsuta, M.: Static analysis of multi-staged programs via unstaging translation. In: 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 81–92. ACM, New York (2011)
8. Cohen, A., Donadio, S., Garzaran, M.J., Herrmann, C., Kiselyov, O., Padua, D.: In search of a program generator to implement generic transformations for high-performance computing. Sci. Comput. Program. 62(1), 25–46 (2006)
9. Dybvig, R.K.: Writing hygienic macros in scheme with syntax-case. Tech. Rep. TR356, Indiana University Computer Science Department (1992)
10. Gordon, A.D.: Bisimilarity as a theory of functional programming. Theoretical Computer Science 228(1-2), 5–47 (1999)
11. Heizmann, M., Jones, N., Podelski, A.: Size-Change Termination and Transition Invariants. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 22–50. Springer, Heidelberg (2010)
12. Herrmann, C.A., Langhammer, T.: Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. Sci. Comput. Program. 62(1), 47–65 (2006)
13. Howe, D.J.: Proving congruence of bisimulation in functional programming languages. Inf. Comput. 124(2), 103–112 (1996)
14. Inoue, J., Taha, W.: Reasoning about multi-stage programs. Tech. rep., Rice University Computer Science Department (October 2011)
15. Intrigila, B., Statman, R.: The Omega Rule is $\mathbf{\Pi^1_1}$-Complete in the $\lambda\beta$-Calculus. In: Ronchi Della Rocca, S. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 178–193. Springer, Heidelberg (2007)
16. Kameyama, Y., Kiselyov, O., Shan, C.-C.: Shifting the stage: Staging with delimited control. In: 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, pp. 111–120. ACM, New York (2009)

17. Kim, I.S., Yi, K., Calcagno, C.: A polymorphic modal type system for LISP-like multi-staged languages. In: 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 257–268. ACM, New York (2006)
18. Kiselyov, O., Swadi, K.N., Taha, W.: A methodology for generating verified combinatorial circuits. In: Proc. of EMSOFT, pp. 249–258. ACM (2004)
19. Koutavas, V., Wand, M.: Small bisimulations for reasoning about higher-order imperative programs. In: 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, pp. 141–152. ACM, New York (2006)
20. Muller, R.: M-LISP: A representation-independent dialect of LISP with reduction semantics. ACM Trans. Program. Lang. Syst., 589–616 (1992)
21. Plotkin, G.D.: The λ-calculus is ω-incomplete. J. Symb. Logic, 313–317 (June 1974)
22. Plotkin, G.D.: Call-by-name, call-by-value and the λ-calculus. Theor. Comput. Sci. 1(2), 125–159 (1975)
23. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74 (2002)
24. Rompf, T., Odersky, M.: Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In: 9th International Conference on Generative Programming and Component Engineering (2010)
25. Sands, D.: Improvement Theory and its Applications, pp. 275–306. Cambridge University Press, New York (1998)
26. Swadi, K., Taha, W., Kiselyov, O., Pašalić, E.: A monadic approach for avoiding code duplication when staging memoized functions. In: 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, pp. 160–169. ACM, New York (2006)
27. Taha, W.: Multistage Programming: Its Theory and Applications. Ph.D. thesis, Oregon Graduate Institute (1999)
28. Taha, W., Nielsen, M.F.: Environment classifiers. In: 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 26–37. ACM, New York (2003)
29. Tsukada, T., Igarashi, A.: A Logical Foundation for Environment Classifiers. In: Curien, P.-L. (ed.) TLCA 2009. LNCS, vol. 5608, pp. 341–355. Springer, Heidelberg (2009)
30. Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., Taha, W.: Mint: Java multi-stage programming using weak separability. In: 2010 Conference on Programming Language Design and Implementation (2010)
31. Yang, Z.: Reasoning about code-generation in two-level languages. Tech. Rep. RS-00-46, BRICS (2000)
32. Yuse, Y., Igarashi, A.: A modal type system for multi-level generating extensions with persistent code. In: 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, pp. 201–212. ACM, New York (2006)