

# Reasoning About Staged Programs

Jun Inoue    Walid Taha    Edwin Westbrook

Dept. of Computer Science, Rice University, Houston, TX, USA  
{ji2,taha,emw4}@rice.edu

## Abstract

Staging is a method for building modular but nevertheless efficient executable specifications in a wide range of applications, including programming language interpreters, highly parametrized numerical code (such as Gaussian elimination) and DSP components (such as FFT). A key feature of this method is that it allows programmers to reduce the run-time penalty for modularity without sacrificing ease of formal reasoning. Unfortunately, there is a scarcity of results and methods rigorously demonstrating this benefit.

This paper addresses two related instances of this problem, namely the absence of a sound equational theory for a CBV staged calculus, as well as a formal analysis of the increasingly important technique of monadic staging. Attaining these results requires revisiting the question of what constitutes a value in a CBV calculus. It also provides a clearer picture about the effects of adding (or removing) staging annotations to (from) a program, and it calls attention to the care needed when using let-insertion to avoid code duplication in staged programs, which appears to have been largely overlooked previously.

**Categories and Subject Descriptors** D.2.4 [Software Engineering]: Software/Program Verification—Correctness Proofs; D.3.1 [Programming Languages]: Formal Definitions and Theory—Semantics; D.3.2 [Programming Languages]: Language Constructs and Features; F.3.0 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs

**General Terms** Languages, Theory, Verification

**Keywords** Staging, Multi-stage programming, Formal verification, Formal semantics, Equational reasoning, Program transformation

## 1. Introduction

High-level abstraction mechanisms give clean and reusable code, but they are often avoided in practice because of the large overhead. An attractive solution to this problem is code generation using multi-stage programming [25], or MSP. Instead of directly running a high-level program, one can annotate the program and make it into a code generator that produces the same program but with abstractions removed. Since the abstraction costs are paid once and for all at generation time, the programmer can aggressively use

abstraction mechanisms without worrying much about the impact on performance.

A rich and diverse set of application domains has arisen for MSP, which includes verified circuits [17], unifying seemingly unrelated techniques for optimal DSP code generation [18], type-safe macros [27], highly parametrized numerical computation [7], efficient programming language interpreters [3, 8], and dynamic programming [24]. A stated benefit of staging is that it makes the programs more amenable to formal analysis in such diverse fields. Unfortunately, very few works try to rigorously demonstrate this point, especially under a general setting. The question of whether the staging really did leave the program's semantics intact is often left unaddressed or only discussed informally.

Brady and Hammond [3] are exceptional in this regard, in that they focus on formally verifying the staged interpreter they obtain in the end. However, their approach is highly dependent upon the specifics of a highly advanced, complex type system, and it may be difficult to transfer to other settings. Furthermore, it does not give intuitive insights as to how staging affects the correctness in general. In contrast, Taha and Johann [27] show how the equational theory of MSP transfers to their typed macro system, although they do not show how it can be used.

Upon closer look, we notice that while there has been significant recent work on multi-stage calculi and related type theories [11, 16, 33, 15, 30], little attention has been given to the equational properties of such languages. Equational reasoning is an excellent match for staged programs for the following reason. A staged program is usually an inefficient but simple, easy-to-analyze code that is annotated to improve performance. The most natural way to verify such a program is to prove that the staged version behaves identically to the unstaged, easy-to-analyze version that we had in the beginning, thus reducing the correctness of the former to that of the latter. The unstaged version can be checked with all of the verification tools and techniques that are developed for single-stage programs, or it can even be correct by construction, as some of the above works demonstrate.

This paper addresses this problem by first presenting an axiomatic equational semantics for a call-by-value (CBV) multi-stage calculus, then showing a substantial case study of verifying a staging method that we call monadic staging [24].

The multi-stage calculus and its basic metatheoretical properties presented here have been investigated in the past [25], but only in the call-by-name (CBN) setting. It is important to have a CBV version of the theory because the vast majority of MSP implementations used and studied today are CBV. We find an obscure caveat in the equational theory, namely that a term of the form  $(\lambda x.e) y$  cannot be equated with  $[y/x]e$  unlike in the single-stage setting, *even when  $e$  does not contain staging constructs*. We demonstrate how the equational theory can be used to rigorously prove that staging annotations did not alter the semantics of a program. The reasoning principles shown here apply broadly to almost any staging strategy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL 2010 January 20-22, 2010, Madrid, Spain.  
Copyright © 2010 ACM [to be supplied]...\$10.00

We observe that this is not enough to do an end-to-end verification of staging because some auxiliary transformations are usually required to write efficient generators. As a substantial case study, we analyze the monadic staging method by Swadi et al. [24], showing the proof obligations and subtleties it involves. This method is particularly important because it is generally applicable and is simple enough for non-experts to understand and apply.

Proofs are only sketched in this paper. Detailed proofs can be found in our companion report [13].

## 1.1 Contributions

MSP is briefly reviewed in Section 2 along with an informal explanation and motivation of monadic staging [24]. A core CBV calculus,  $\lambda_v^U$ , is presented in Section 3. This calculus and its basic properties were presented before in CBN [25] but not in CBV. We discuss in Section 3.3 why  $\beta$  substitution is unsound if the argument is a variable, and why this restriction only applies to the untyped setting. In Section 4, we show that if a staged program is provably equal to *some* unstaged term according to this calculus, then it is equivalent to its unstaged counterpart. Our analysis up to this point is completely independent of any type system.

In Section 5, we precisely codify the monadic memoizing CPS transformation, which is the main tool in monadic staging that eliminates code duplication. We show that it is semantics-preserving, provided the transformed program satisfies a simple well-typedness condition.

In Section 6, we conclude our case study by describing the proof obligations needed to justify `let`-insertion. While this seems like simple  $\beta$  expansion, it turns out to be the most problematic part of monadic staging, due to a problem caused by the code motion it entails. The same problem was already known to Bondorf [2], but it was apparently overlooked when his method was streamlined and transferred to MSP. The problem did not surface in the studies of MSP by Swadi et al. [24] or Carrette [7], but it poses a significant obstacle to applying this to more complex, feature-hungry applications. This pitfall makes a perfect case for our study—it is critical to have rigorous studies of correctness like ours in order to prevent such caveats from going unnoticed.

## 2. Staging and Monadic Staging

We begin with a brief review of staging features in the MSP language MetaOCaml [26]. All examples are written in this language. We then summarize the monadic staging technique, which will be formalized in Section 5.

### 2.1 Staging Basics

MetaOCaml is an extension of OCaml [19] with three language constructs: brackets (`.<_>.`), escape (`.~`), and run (`.!`). While MetaOCaml is impurely functional, we will only consider its purely functional subset.

Brackets generate code. For example,

```
.<(fun x → x * x) 2>.
```

evaluates to the parse tree of `((fun x → x*x) 2)`. The type of this value is `<int>`, read “code of `int`”.<sup>1</sup> Just like quasiquotation in LISP-like languages, `.<_>.` gives a syntax for writing down code as a literal value.

Escapes allow parts of generated code to be computed dynamically, similar to LISP `unquote`. For example:

```
let add1 = .<.~x + 1>. in
```

<sup>1</sup> Actually, MetaOCaml assigns it the type `(’a, int) code` where `’a` is an environment classifier [28]. Classifiers are orthogonal to the results presented here and are therefore omitted in all types.

```
add1 .<2 * 3>.
```

When `add1` is called, `.<2*3>.` gets substituted for `x` and we get the intermediate term `.<.~(.<2+3>.) + 1>.` Then `.~` merges the code value `.<2*3>.` into the surrounding, which gives `.<(2*3) + 1>.` as the final value. The `x` inside `.~` can be replaced by any expression that returns a code value of the right type. `.~` can only appear (lexically) inside `.<_>.`

`.<_>.` and `.~` can be nested, but `.~` cannot be nested deeper than `.<_>.` The depth of `.<_>.` minus the depth of `.~` is called the *level*. Execution of escaped code is deferred like its surrounding if the nesting of `.<_>.` is strictly deeper (i.e. `level > 0`).

Finally, we have an analogue of LISP’s `eval` function, which is `run (. !)`. This construct takes a code value, compiles it into machine code (or byte code, depending upon the flavor of MetaOCaml being used), which is immediately executed. For example,

```
.! .<(fun x → x * x) 2>.
```

returns 4.

The advantages of staging annotations over its LISP counterparts is automatic hygiene [6] (like Scheme macros [10]) but unlike traditional LISP macros), support for equational reasoning within quoted parts of the program text [25], and type safety in the absence of side effects [28].

Resolving escapes and generating a code value is called stage 0, while executing the generated code with `.!` is referred to as stage 1. Stage 1 code may also create and run another code value, which would constitute stage 2, but this is rarely useful.

The use of the `+` operator inside brackets is an example of cross-stage persistence (CSP). In general, values bound in stage 0 can be used in stage 1 (and later stages) as constants. This feature is more subtle than it may seem at a glance, and it poses problems for monadic staging (see Section 2.2).

### 2.2 Monadic Staging

Let us show an example of staging “in action,” which we will use throughout the paper to illustrate our reasoning principles. Figure 1 shows unstaged and staged implementations of the Generalized Fibonacci (Gibonacci) sequence. Gibonacci is a sequence taking a pair of numbers `x` and `y` as parameters, and it is defined by the same recurrence as the Fibonacci sequence but starts out as “`x, y, x+y, ...`” instead of “`0, 1, 1, ...`”.

`gib` is the unstaged version, and it is just a transcription of the definition into OCaml code. Adding staging annotations to `gib` gives the code generator `gibgen`. When invoked like

```
gibgen 4 .<x> .<y>.
```

it produces a code value representing the computation that would commence when `gib 4 x y` is called, except with all the recursion unrolled and branching eliminated.

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

Notice, however, that `gibgen` suffers from the *code duplication problem*, which is among the most common pitfalls of staging. The subexpression `y+x` is repeated twice in the generated code, and a similar redundancy is introduced every time we call `gibgen` with

```

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

(* Staged *)
(* gibgen : int → <int> → <int> → <int> *)
let rec gibgen n x y =
  if n = 0 then x else
  if n = 1 then y else
  let r1 = gibgen (n-1) x y
  and r2 = gibgen (n-2) x y in
  .<~r1 + ~r2>.

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

(* Monadically staged *)
(* gibgenm : int → <int> → <int> → <int> *)
let gibgenm n x y =
  let gibgenm' f n x y =
    if n = 0 then ret x else
    if n = 1 then ret y else
    bind (f (n-1) x y) (fun r1 →
    bind (f (n-2) x y) (fun r2 →
    ret .<~r1 + ~r2>))
  in
  mfixl gibgenm' n x y empty kval

(* execution examples *)
gibst 0 2 3 ⇒ 2
gibst 4 2 3 ⇒ 8
(* assuming x,y are bound: *)
gibgen 0 .<x>. .<y>. ⇒ .<x>.
gibgen 4 .<x>. .<y>. ⇒ .<(((y+x)+y)+(y+x))>.
gibgenm (4, .<x>., .<y>.) empty kval
⇒
.<let z_3 = y in
  let z_4 = x in
  let z_5 = (z_3 + z_4) in
  let z_6 = (z_5 + z_3) in
  let z_7 = (z_6 + z_5) in z_7>.

```

Figure 1. Staging Gibonacci.

a given argument for the second (or later) time. This is because `gib` is an inefficient implementation that incurs many redundant calculations, while `gibgen` faithfully unrolls the exponential amount of computation into a code value.

`gibgenm` is the result of applying the monadic memoization technique to `gib`. It uses a state-continuation monad to carry around the memo table, and a special memoizing fixed-point operator named `mfixl`, defined in Figure 2. The monad is not just a state monad but also supplies continuations for the following reason.

Consider what happens when we try to improve efficiency by naively memoizing `gib` and `gibgen` so that we literally preserve the return values of these functions. `gib` becomes linear-time, but

```

(* Monadic operations (simplified2) *)
let ret x = fun s k -> k s x
and bind m f = fun s k ->
  m s (fun s' r -> f r s' k)
and kval _ v = v
and kst s _ = s

(* Memoizing monadic fixed-point *)
let rec mfixl f n x y =
  fun t k ->
  match lookup (n, x, y) t with
  | Some v -> ret v
  | None ->
    (f (mfixl f) n x y) (fun t' c ->
    .<let z = ~c in
    ~.(k (add (n,x,y, .<z>.) t') .<z>.)>.)

```

Figure 2. Auxiliary definitions for Gibonacci example.

`gibgen` will generate the same exponential code. The unmemoized function returns an exponentially large code value, so the memoized version does as well.

Upon a closer look at this problem, we notice that whereas a memoized version of `gib` would insert the *result* of, say, `x+y` to the memo table, the naively memoized version of `gibgen` would insert `.<x+y>.`, which is a representation of the *work* of `x+y`. Then whenever we copy this entry, we would be copying the work. We want to populate the table instead with entries of the form `.<z>.`—i.e. a variable that is bound *in stage 1* to the result of `x+y`. Such an entry embodies zero work, so there is no harm in replicating them. However, this raises a difficult question: where is the `z` going to be bound?

The `z` has to be bound in (the memoized version of) `gibgen`, because the caller has no idea how many distinct `z`'s are needed. But we cannot just return an open term `.<z>.` without returning the binding with it. This is an essential safety feature that is bolted into MSP semantics (see Proposition 3). So, when a cache is found in the table, the memoized function is forced to return the binding along with the `z`, like `.<let z = x + y in z>.` But we would then be replicating the `x+y` again. Hence we have a dilemma: we want to return `.<z>.`, but we don't want to return control to the caller because that would force us to insert a `let`.

CPS transformation is a standard solution to this dilemma. In CPS, we no longer return to the caller but call it back, so that we can just pass in `.<z>.` without supplying the binder. When the callback finishes, we wrap the whole thing in the context `.<let z = [] in ...>.` and obtain a closed code value. This machinery is implemented in `mfixl`. We take table-manipulating functions `lookup` and `add` for granted. `empty` is the empty table and `kval` is the continuation that extracts the return value while throwing away the updated state. `kst` is the dual continuation that throws away the return value and gets the state.

CPS transformation is, however, menial and hard to get right by hand. Programmers tend to make small mistakes that are, although often caught by the type system, hard to diagnose and fix. The monadic staging method proposed by Swadi et al. [24] encapsulates this into a monad, which also happens to free us from the burden of explicitly carrying around the memo table. Memoized staging can be summed up as follows:

<sup>2</sup>`bind` requires rank-2 polymorphism and is not typable in MetaOCaml as written in Figure 2. It can be typed using MetaOCaml's first-class polymorphism feature, but we did not do that here to avoid the notational overhead.

- (i) Add staging annotations to create a code generator.
- (ii) Monadify the generator and use open recursion.
- (iii) Close the recursion with `mfixl`.

Note that monadic staging is not only good for a memoized program, although that is where it provides the most dramatic improvements. Monadic staging essentially does common subexpression elimination (CSE) on the generated code, which is a useful thing under most circumstances. Techniques for making the CSE finer-grained are presented in [7] and [24].

### 3. The $\lambda_v^U$ Calculus

As with any practical language, MetaOCaml is too large to use in mathematical analyses. We present here a core calculus,  $\lambda_v^U$ , that models the untyped, purely functional subset of MetaOCaml. We give a big-step semantics and an axiomatic equational theory. Provable equality in this theory is an approximation of observational equivalence, which is strong enough to equate all terms that terminate to a given value.

$\lambda_v^U$  is a CBV variant of  $\lambda^U$  [25], which is CBN. Much of the metatheoretical results presented here (confluence and equivalence of the different forms of semantics) have already been established for  $\lambda^U$ , but they had not been transferred to the CBV setting, which is better suited for MetaOCaml and most other existing MSP language implementations. Complete proofs may be found in the technical report [13].

There are several typed formalisms available for modeling MSP [5, 33, 15], but we deliberately use an untyped formalism because it allows us to focus on the operational properties of MSP and establish results without committing to a particular type system.

There are two main differences between CBV and CBN. One is the restriction (in CBV) of  $\beta$  substitution to value-arguments, which is the defining characteristic of CBV. The other is the treatment of variables. In single-stage calculi, one can define a variable to be a value and apply  $\beta$  reduction to  $(\lambda x.e) y$ . We find that it is not justifiable for multi-level languages in the untyped setting.

#### 3.1 Syntax and Natural Semantics

Figure 3 shows the syntax of  $\lambda_v^U$ . It is mostly identical to that of  $\lambda^U$ , having variables, abstractions, applications, brackets, escapes, and run. An incomplete expression  $C$  with exactly one hole  $\square$  is called a context.  $C[e]$  is the complete expression obtained by replacing the hole with  $e$ . We informally refer to a “level  $n$  context”, which is a context whose hole is enclosed by  $n$  uncanceled brackets, where an escape cancels one enclosing pair of brackets.

##### Expressions

$$e \in E ::= x \mid \lambda x.e \mid e e \mid \langle e \rangle \mid \sim e \mid \text{run } e$$

##### Stratified Expressions

$$\begin{aligned} e^0 \in E^0 &::= x \mid \lambda x.e^0 \mid e^0 e^0 \mid \langle e^1 \rangle \mid \text{run } e^0 \\ e^{n+1} \in E^{n+1} &::= x \mid \lambda x.e^{n+1} \mid e^{n+1} e^{n+1} \\ &\quad \mid \langle e^{n+2} \rangle \mid \sim e^n \mid \text{run } e^{n+1} \end{aligned}$$

##### Values

$$\begin{aligned} v^0 \in V^0 &::= \lambda x.e^0 \mid \langle e^0 \rangle \\ v^{n+1} \in V^{n+1} &::= e^n \end{aligned}$$

##### Contexts

$$C \in Ctx ::= \square \mid \lambda x.C \mid C e \mid e C \mid \langle C \rangle \mid \sim C \mid \text{run } C$$

Figure 3.  $\lambda_v^U$  syntax.

**Notation.** We use  $(\equiv)$  for  $\alpha$  equivalence.  $\lambda x_1 x_2 \dots x_n.e$  is shorthand for  $\lambda x_1.\lambda x_2.\dots \lambda x_n.e$ . In definitions, we may write  $f x \stackrel{\text{def}}{=} e$  instead of  $f \stackrel{\text{def}}{=} \lambda x.e$ .

Expressions are stratified into levels. Informally, an expression is at level  $n$  if its nesting of escapes is no more than  $n$  levels deeper than brackets at any point in the expression. A legal program is a closed level 0 expression, and a level  $n$  subexpression that does not also belong to a lower level can appear only in a level  $n$  context. Values at level  $n$  are expressions that contain no more work at that level, which for  $n = 0$  is an abstraction or a code value with no unresolved escapes, and for  $n > 0$  is any lower-level expression.

Figure 4 shows the natural (big-step) semantics for  $\lambda_v^U$ . The big-step rules are also very similar to those of  $\lambda^U$ , except that  $\beta$  substitution requires the argument to be a (level-0) value. There is a big-step relation at each level, which is a partial function because the rule that can derive a judgment  $e_1^n \xrightarrow{n} e_2^n$  is determined uniquely by  $n$  and  $e_1^n$ .

The most important correspondence between the syntax and the big-step semantics is the fact that  $(\xrightarrow{n})$  picks out a unique level- $n$  value (or none) for each  $e^n$  that it terminates to, and that every value terminates to itself. This is our justification for the syntactic definition of values.

**Proposition 1.** *It is the case that:*

- (i)  $e_1^n \xrightarrow{n} e_2^n \implies e_2^n \in V^n$ .
- (ii)  $v^n \xrightarrow{n} v^n$  for any  $v^n$ .

This justifies the following definition.

**Definition 2 (Termination and Evaluation).** We can define a partial function  $\text{Eval}_n : E^n \rightarrow V^n$ , set to return the unique  $v^n$  such that  $e^n \xrightarrow{n} v^n$ . When  $\text{Eval}_n(e)$  is defined, we say that  $e$  terminates (at level  $n$ ), written  $e \Downarrow^n$ . Otherwise,  $e$  is said to diverge.

$\lambda_v^U$  semantics does not permit scope extrusion, meaning that variables (especially future-stage variables) do not escape the scope of their binders and become free, unless they were free from the beginning. This is an essential safety feature that we alluded to in Section 2.2.

**Proposition 3.** *Let  $FV(e)$  be the set of free variables in  $e$ . Then  $e^n \xrightarrow{n} v^n \implies FV(e^n) \supseteq FV(v^n)$ . In particular, a function cannot return a code value containing free variables, except those that were already free to begin with.*

The most intuitive notion of equivalence for two program fragments, which we care about most, is that of substitutability—replacing a subexpression with an equivalent one does not alter the observable behavior of the whole program. In other words, it is a congruence that preserves the observable behavior. Since  $\lambda_v^U$  is untyped, we restrict the notion of observation to termination at level 0.

**Definition 4 (Observational Equivalence).**

$$\begin{aligned} e_1 \approx e_2 &\stackrel{\text{def}}{\iff} \\ &\forall C. (C[e_1], C[e_2] \in E^0 \implies (C[e_1] \Downarrow^0 \iff C[e_2] \Downarrow^0)) \end{aligned}$$

One can easily check that  $(\approx)$  is indeed a congruence.

#### 3.2 Equational Theory and Reduction Semantics

Observational equivalence is difficult to reason about directly. It is not semi-decidable and showing an equivalence directly from the definition usually requires prohibitive amounts of effort. An important advantage of MSP is the presence of a strong, axiomatic equational theory that can simplify this task. This theory provides an

$$\begin{array}{c}
\frac{}{x \stackrel{n+1}{\hookrightarrow} x} \text{ [BS-VAR]} \quad \frac{}{\lambda x.e^0 \stackrel{0}{\hookrightarrow} \lambda x.e^0} \text{ [BS-ABS0]} \quad \frac{e_1^{n+1} \stackrel{n+1}{\hookrightarrow} e_2^{n+1}}{\lambda x.e_1^{n+1} \stackrel{n+1}{\hookrightarrow} \lambda x.e_2^{n+1}} \text{ [BS-ABS+]} \quad \frac{e_1^0 \stackrel{0}{\hookrightarrow} \lambda x.e^0 \quad e_2^0 \stackrel{0}{\hookrightarrow} v_2^0}{[v_2^0/x]e^0 \stackrel{0}{\hookrightarrow} v^0} \text{ [BS-APP0]} \\
\frac{e_{f_1}^{n+1} \stackrel{n+1}{\hookrightarrow} e_{f_2}^{n+1} \quad e_{a_1}^{n+1} \stackrel{n+1}{\hookrightarrow} e_{a_2}^{n+1}}{e_{f_1}^{n+1} e_{a_1}^{n+1} \stackrel{n+1}{\hookrightarrow} e_{f_2}^{n+1} e_{a_2}^{n+1}} \text{ [BS-APP+]} \quad \frac{e_1^{n+1} \stackrel{n+1}{\hookrightarrow} e_2^{n+1}}{\langle e_1^{n+1} \rangle \stackrel{n}{\hookrightarrow} \langle e_2^{n+1} \rangle} \text{ [BS-BRK]} \quad \frac{e_1^0 \stackrel{0}{\hookrightarrow} \langle e_2^0 \rangle}{\sim e_1^0 \stackrel{1}{\hookrightarrow} e_2^0} \text{ [BS-ESC1]} \\
\frac{e_1^{n+1} \stackrel{n+1}{\hookrightarrow} e_2^{n+1}}{\sim e_1^{n+1} \stackrel{n+2}{\hookrightarrow} \sim e_2^{n+1}} \text{ [BS-ESC+2]} \quad \frac{e_1^0 \stackrel{0}{\hookrightarrow} \langle e_2^0 \rangle \quad e_2^0 \stackrel{0}{\hookrightarrow} e_3^0}{\text{run } e_1^0 \stackrel{0}{\hookrightarrow} e_3^0} \text{ [BS-RUN0]} \quad \frac{e_1^{n+1} \stackrel{n+1}{\hookrightarrow} e_2^{n+1}}{\text{run } e_1^{n+1} \stackrel{n+1}{\hookrightarrow} \text{run } e_2^{n+1}} \text{ [BS-RUN+]}
\end{array}$$

Figure 4. Big-step semantics for  $\lambda_v^U$ .

inductively defined provable equality that effectively approximates ( $\approx$ ).

Figure 5 shows the axioms for provable equality. ( $=$ ) is also clearly a congruence.

$$\begin{array}{c}
\frac{}{(\lambda x.e^0) v^0 = [v^0/x]e^0} [\beta_v] \quad \frac{}{\sim \langle e^0 \rangle = e^0} [E_U] \\
\frac{}{\text{run } \langle e^0 \rangle = e^0} [R_U] \quad \frac{}{e = e} [\text{EQ-REFL}] \\
\frac{e_1 = e_2 \quad e_2 = e_3}{e_1 = e_3} [\text{EQ-TRAN}] \quad \frac{e_1 = e_2}{e_2 = e_1} [\text{EQ-SYM}] \\
\frac{e = e'}{C[e] = C[e']} [\text{EQ-CTX}]
\end{array}$$

Figure 5. Provable equality for  $\lambda_v^U$ .

The central result that relates the axiomatic semantics to the big-step semantics is Theorem 5, stating that the two are equivalent: the equational theory is able to discover all terminating expressions and the values they terminate to.

**Theorem 5** (Equivalence of Axiomatic and Big-step Semantics). *It is the case that:*

- (i)  $e^n = v_1^n \implies \exists v_2^n. e^n \stackrel{n}{\hookrightarrow} v_2^n = v_1^n$
- (ii)  $e^n \stackrel{n}{\hookrightarrow} v^n \implies e^n = v^n$

The relevance of provable equality to formal reasoning for MSP rests on the fact that it approximates ( $\approx$ ):

**Corollary 6** (Soundness and Incompleteness).  $(=) \subset (\approx)$ .

A sound axiomatic theory for ( $\approx$ ) is generally incomplete, because an inductively defined relation is usually semi-decidable whereas ( $\approx$ ) is not (since  $\lambda_v^U$  is Turing-complete). It may be possible to write derivation rules that break semi-decidability and get the theory closer to being complete (say, by allowing quantifiers in the rules), but it would no longer really be an axiomatic theory and we would lose much of the ease of use.

The main inconvenience in proving Theorem 5 is the fact that ( $=$ ) has no sense of direction of execution. For example, if the  $\beta_v$  rule is used to rewrite a term from left to right, then intuitively it should get us “closer” to termination, while a right-to-left rewrite can be seen as rewinding an execution, getting “farther away” from termination. The equational theory cannot distinguish between these notions because of the EQ-SYM rule. It helps in proving Theorem 5

and doing other meta-reasoning about the equational theory to define a “directed” version of ( $=$ ) that recovers the notion of direction of execution while still being a precongruence.

Figure 6 defines the reduction semantics. We write ( $\longrightarrow^*$ ) for the reflexive-transitive closure of ( $\longrightarrow$ ), and ( $\longrightarrow^+$ ) for the transitive closure of ( $\longrightarrow$ ). We write  $e_0 \longrightarrow^k e_k$  to mean there is a length- $k$  chain  $e_0 \longrightarrow e_1 \longrightarrow \dots \longrightarrow e_k$ . The ( $\longrightarrow^*$ ) is the precongruence that we just mentioned.

A reduction that doesn’t use the RED-CTX rule is called a *primitive reduction*. One can easily see that if  $e \longrightarrow e'$ , then there are (not necessarily unique)  $C$ ,  $e_1$ , and  $e_2$  such that  $e \equiv C[e_1] \longrightarrow C[e_2] \equiv e'$ , and  $e_1 \longrightarrow e_2$  is a primitive reduction. This  $C$  is called the *reduction context*.

$$\begin{array}{c}
\frac{}{(\lambda x.e^0) v^0 \longrightarrow [v^0/x]e^0} [\beta_v] \quad \frac{}{\sim \langle e^0 \rangle \longrightarrow e^0} [E_U] \\
\frac{}{\text{run } \langle e^0 \rangle \longrightarrow e^0} [R_U] \quad \frac{e \longrightarrow e'}{C[e] \longrightarrow C[e']} [\text{RED-CTX}]
\end{array}$$

Figure 6. Reduction semantics for  $\lambda_v^U$ .

**Remark.** Note the difference from a small-step semantics, which specifies a unique redex to be reduced, and which generally does not go under  $\lambda$  to find it. The reduction semantics does not have either property, as it is a retake of the equational theory and not a model of the execution trace of a program.

It is immediately obvious that ( $\longrightarrow^*$ )  $\subset$  ( $=$ ). More precisely, ( $=$ ) is the symmetric-transitive closure of ( $\longrightarrow^*$ ), or equivalently, the smallest congruence containing ( $\longrightarrow$ ) or ( $\longrightarrow^*$ ).

Some key properties of ( $=$ ) and ( $\longrightarrow^*$ ) follow from confluence, or the Church-Rosser property.

**Theorem 7** (Confluence). *If  $e_1 \longleftarrow^* e \longrightarrow^* e_2$ , then  $\exists e'. e_1 \longrightarrow^* e' \longleftarrow^* e_2$ .*

A key consequence is that expressions are provably equal iff they have a common reduct.

**Corollary 8.**  $e_1 = e_2 \iff \exists e_3. e_1 \longrightarrow^* e_3 \longleftarrow^* e_2$ .

Informally,  $e_1 = e_2$  holds when there is a sequence that looks like

$$e_1 \longrightarrow^* e_3 \longleftarrow^* e_4 \longrightarrow^* \dots \longleftarrow^* e_{k-1} \longrightarrow^* e_k \longleftarrow^* e_2.$$

Confluence finds a common reduct for neighboring expressions in this sequence, and also for the common reducts themselves. Repeat-

ing the process, we eventually converge on a single expression to which all the  $e_i$  reduce.

Using this fact, we can derive the equivalence between ( $=$ ) and ( $\longrightarrow^*$ ) in terms of the ability to discover terminating expressions.

**Corollary 9.**  $e^n = v_1^n \implies \exists v_2^n. e^n \longrightarrow^* v_2^n = v_1^n$ .

*Proof.* The common reduct of  $e^n$  and  $v_1^n$  assured by Corollary 8 is a level- $n$  value (because it's a reduct of a value,  $v_1^n$ —see the technical report [13] for a lemma proving this formally).  $\square$

Confluence is essential to the proofs of both corollaries. By Corollary 9, Theorem 5 follows directly from the following theorem, which is more natural to prove than Theorem 5.

**Theorem 10.** *The big-step semantics is equivalent to the reduction semantics:*

- (i)  $e^n \xrightarrow{n} v^n \implies e^n \longrightarrow^* v^n$
- (ii)  $e^n \longrightarrow^* v_1^n \implies \exists v_2^n. e^n \xrightarrow{n} v_2^n \longrightarrow^* v_1^n$ .

Theorems 7 (confluence) and 10 (equivalence of big-step and axiomatic semantics) remain unproved. Proofs of these theorems are nearly identical to Taha's [25], which in turn imitates approaches by Plotkin [23] and Takahashi [29]. See the technical report [13] for details.

### 3.3 Variables Are Not Values

$\lambda_v^U$  has an interesting point of difference from other CBV lambda calculi in the way it treats variables. There is a tradition going back to Plotkin [23] to admit the equational rule

$$\frac{}{(\lambda x. e^0) y = [y/x]e^0} \text{ [EQ-VAR}\beta\text{]},$$

in CBV calculi. Plotkin, for example, does this implicitly by taking variables to be (level-0) values, subsuming it by the  $\beta_v$  rule. This rule is indispensable for abstract interpretation, for example contracting the application in ( $\mathbf{fun} \ x \rightarrow \mathbf{id} \ x$ ), where  $\mathbf{id}$  is the identity function.

This rule, however, is *not* admissible in  $\lambda_v^U$  unless we add context-sensitive well-formedness conditions (i.e., a type system). The following code illustrates this point:

```
.<fun x → .~(let _ = (fun _ → 0) x in .<0>.>.>.
```

This is clearly ill-leveled—we bind  $x$  at level 1 and use it at level 0—but is a legitimate  $\lambda_v^U$  expression nonetheless. The  $x$  in the application does not (big) step to a value because there is no rule for variables at level 0, so the code inside escapes does not terminate, and neither does the entire code. However, if we allow EQ-VAR $\beta$ , this term would be provably equal to  $\mathbf{fun} \ x \rightarrow 0$ , a value. Note that the EQ-VAR $\beta$  “redex” in question here is  $(\mathbf{fun} \ _ \rightarrow 0) \ x$ , which does not contain staging annotations at all.

There are two conceivable rules for level-0 variables,

$$\frac{}{x \xrightarrow{0} x} \text{ [OBS-VAR0]} \quad \text{and} \quad \frac{}{x \xrightarrow{0} \mathbf{err}} \text{ [CBS-VAR0]}.$$

As the reader is aware, our intention in Figure 4 was to have CBS-VAR0 but to omit it (along with other error-generation rules) from the presentation since we are not proving type safety. Let us call a semantics with OBS-VAR0 an open-term big-step semantics (OBS) and one with CBS-VAR0 a closed-term big-step semantics (CBS).

On the one hand, CBS is the “correct” semantics because it is the one that compilers and interpreters implement. However, it makes the EQ-VAR $\beta$  rule unsound. That is to say, Theorem 5 becomes false if we define  $x \in V^0$ . On the other hand, OBS is compatible with EQ-VAR $\beta$  (allows  $x \in V^0$ ) but deviates from the actual implementation's behavior.

This difference is immaterial in single-stage calculi because CBS and OBS coincide if we restrict observation to closed terms (as is usually done)—in other words, if we define  $e_1 \approx e_2$  iff  $C[e_1] \Downarrow^0 \iff C[e_2] \Downarrow^0$  for any  $C$  such that the  $C[e_i]$  are *closed* and at level 0. The rules that exist in single-stage calculi (namely BS-ABS0 and BS-APP0) never go under lambda, and the set of allowed free variables does not grow as we traverse the proof tree upward. Thus, if we start out asking the value (or termination) of a closed term, we never have to ask the value of an open term, so OBS-VAR0 and CBS-VAR0 are dead rules. Hence, it does not matter which rule we choose, and by extension whether or not we include EQ-VAR $\beta$ . In  $\lambda_v^U$ , BS-ABS+ breaks this property so that OBS and CBS do not coincide. This happens independently of the choice about observing open terms. Hence we are forced to exclude EQ-VAR $\beta$  from  $\lambda_v^U$ .

Interestingly enough, this problem goes away, and we can recover EQ-VAR $\beta$  if we introduce some form of type checking. In a type-safe system that observes only closed terms, the CBS-VAR0 and OBS-VAR0 are dead because the type system guarantees that a free variable is never looked up.

We will still stick to the untyped system despite this discussion because reconstructing the results that follow for a system allowing EQ-VAR $\beta$  is trivial, and not having as little types as possible simplifies the presentation. As such, we will not treat variables as values hereafter.

### 3.4 Extensionality

Another very common equational axiom is the extensionality rule,

$$\frac{x \notin FV(\lambda y. e^0)}{\lambda y. e^0 = \lambda x. (\lambda y. e^0) x} \text{ [EQ-}\eta_v\text{]}.$$

This rule is manifestly unsound if we allow the  $\lambda y. e^0$  to be replaced by a code value or a non-value. Even if we restrict our attention to terms that we know are functions, however, this rule is quite useful because it is not uncommon that a pair of functions cannot be (easily) proved equivalent directly but are easily proved equal once we apply them to a common value.

Proving the admissibility of  $\eta_v$  in the absence of EQ-VAR $\beta$  is somewhat involved. One approach is to include it officially as a derivation rule for provable equality (i.e., add it to Figure 5) and show that the resulting equational theory is still equivalent to the big-step semantics, is confluent, etc, and finally that ( $=$ )  $\subseteq$  ( $\approx$ ) still holds. It is also possible to show it directly and to do so for a stronger and often more useful rule:

$$\frac{\forall v^0. [v^0/x]e_1^0 \approx [v^0/x]e_2^0}{e_1^0 = e_2^0} \text{ [EQ-EXT]}.$$

The reader should consult the technical report [13] for a proof.

## 4. The Effects of Staging on Program Behavior

Most verification tools and techniques are designed for single-stage programs, and we cannot expect them to be able to reason about staged programs like  $\mathbf{gibst}$  and  $\mathbf{gibgen}$  directly. In order to use them in this setting, we exploit the fact that the vast majority of staged programs are just annotated variants of some underlying unstaged counterparts, such as  $\mathbf{gib}$  in the Fibonacci example.

In this section, we show (Corollary 13) that a staged program is provably equal to its unstaged counterpart, provided that its staging annotations can be eliminated via reductions and reverse reductions. This reduces the correctness of a staged program to its unstaged counterpart, and allows us to get away by verifying the unstaged one instead.

#### 4.1 Correctness of Staging

**Definition 11** (Erasure). Define a syntactic transformation  $\|\cdot\| : E \rightarrow E$  by

$$\begin{aligned} \|x\| &\stackrel{\text{def}}{=} x, & \|e_1 e_2\| &\stackrel{\text{def}}{=} \|e_1\| \|e_2\|, & \|\lambda x. e\| &\stackrel{\text{def}}{=} \lambda x. \|e\|, \\ \|\langle e \rangle\| &\stackrel{\text{def}}{=} \|e\|, & \|\tilde{e}\| &\stackrel{\text{def}}{=} \|e\|, & \|\text{run } e\| &\stackrel{\text{def}}{=} \|e\| \end{aligned}$$

$\|e\|$  is called the erasure of  $e$ .

The image of  $\|\cdot\|$  is given by the grammar

$$\|e\| ::= x \mid (\lambda x. \|e\|) \mid (\|e\| \|e\|)$$

which is just the annotation-free terms. The “unstaged counterpart” is formally defined as the erasure of the staged program. Our goal then becomes to prove an equivalence of the form  $e \approx \|e\|$ . In general, this can involve a full-blown analysis of the execution of the staged program under arbitrary contexts, but we may get some assistance from the following facts to simplify the proof.

**Proposition 12.** *Erasure preserves reduction:*  $e \rightarrow^* e' \implies \|e\| \rightarrow^* \|e'\|$ .

**Corollary 13** (Correctness Criterion).  $e = \|e\| \iff \exists e'. e = \|e'\|$ .

Proposition 12 gives us some insights about the effects of staging annotations on program behavior. If a program  $e^0$  terminates to a value  $v^0$ , then its erasure  $\|e^0\|$  reduces to  $v^0$  with some staging annotations added, say  $e$ . This  $e$  is not necessarily a value (e.g.  $v^0 = \lambda x.x$  and  $e = \langle \lambda x. \tilde{x} \rangle$ ), but if we can assure that it is, then it follows that  $\|e^0\| \Downarrow^0$  and  $e^0 \approx \|e^0\|$ . This is obvious if, for example, the return type of the staged program is `int` or `float`. Put another way, staging/erasure is always a partially correct transformation modulo staging annotations in the return value.

Corollary 13 is useful for programs that do not necessarily terminate, for example interpreters. To show  $e = \|e\|$ , we only need to find *an* unstaged term  $\|e'\|$  that is equal to  $e$ . Proving  $\|e'\| \equiv \|e\|$  or  $\|e'\| \approx \|e\|$  is not necessary. The non-deterministic nature of the axiomatic/reduction semantics may come in handy here: programmers have more freedom in the way they find a  $\|e'\|$  than if they were forced to follow the deterministic execution.

The condition  $\exists e'. e = \|e'\|$  is nonetheless non-trivial to establish in general, but the example in the next subsection should convince the reader that we can expect the task to be fairly straightforward for a wide variety of programs.

#### 4.2 Example: Verifying Staged Fibonacci

Let us illustrate how Corollary 13 can be used to verify a staged program, using the Fibonacci implementation from Section 2.2.

First, `fib` is correct by construction. This is a major advantage often enjoyed by programs that are not encumbered with performance constraints: it is straightforward enough to be analyzed relatively quickly and easily. Now, we can verify `fibst` by reducing its correctness to that of `fib` using Proposition 13.

**Proposition 14.** *For any non-negative integer  $n$  and integers  $x_0, y_0$ ,*

$$\text{fib } n \ x_0 \ y_0 \approx \text{fibst } n \ x_0 \ y_0$$

**Remark.** Technically, the `fib` and `fibst` are free variables in this equation. We of course mean the `let rec` forms in Figure 1 that bind those names to be part of the equation, but they are omitted for conciseness.

*Proof.* One can easily check that the erasure of the right-hand side (rhs) is provably equal to the left-hand side (lhs). So by Proposition 13 and Corollary 6, we only need to prove that the rhs is provably equal to some  $\|e\|$ . The rhs  $\beta_v$ -reduces to

$$(\cdot! \langle \text{fun } x \ y \rightarrow \cdot \tilde{(\text{gibgen } n \ \langle x \rangle \ \langle y \rangle)} \rangle) \ x_0 \ y_0$$

which reduces to an annotation-free term iff the escaped part reduces to a term of the form  $\langle e^u \rangle$ . We prove this by induction on the termination measure of `gibgen`—i.e.  $n$ . The base cases  $n = 0, 1$  are easy. For the inductive step, IH gives

$$\begin{aligned} \text{gibgen } (n-1) \ \langle x \rangle \ \langle y \rangle &\rightarrow^* \langle e_1^u \rangle. \\ \text{gibgen } (n-2) \ \langle x \rangle \ \langle y \rangle &\rightarrow^* \langle e_2^u \rangle. \end{aligned}$$

from which we get

$$\text{gibgen } n \ \langle x \rangle \ \langle y \rangle \rightarrow^* \langle e_1^u + e_2^u \rangle.$$

which concludes the proof. Note that we do not need to explicitly relate the  $e_i^u$  or  $e_1^u + e_2^u$  to `gib` in any way.  $\square$

As a general strategy, we induct on the termination measure of the generator to prove that code generation terminates with a value of the form  $\langle e^u \rangle$ . We can expect this to work most of the time because generators are usually designed with such an invariant in mind. This is especially true of generators like `gibgen` that started out as an unstaged program and subsequently acquired staging annotations to improve efficiency.

Proving non-termination for negative  $n$  is somewhat harder, having to rely on the deterministic semantics (augmented with error-generating rules) to show that there is no derivation of a big-step judgment.

## 5. Memoizing Monadic Transformation

In this section, we codify the monadic CPSing-memoizing transformation discussed in Section 2.2. We show a pair of algorithmic transformations  $\mathcal{M} : E^0 \rightarrow E^0$  and  $\Psi : V^0 \rightarrow E^0$  for adding monadic memoization.

An important deviation from the description in Section 2.2 is that we separate memoization from `let`-insertion, and have  $\mathcal{M}/\Psi$  do only the memoization part. More precisely, we use a combinator `mfix` in place of `mfixl`: the only difference is that the

$$\begin{aligned} \cdot \text{let } z = \cdot \tilde{c} \text{ in} \\ \cdot \tilde{(k \ (\text{add } (n, x, y, \cdot \langle z \rangle) \ t') \ \langle z \rangle)} \end{aligned}$$

in the code for `mfixl` in Figure 2 is replaced by

$$k \ (\text{add } (n, x, y, c) \ t') \ c.$$

The `let`-insertion is a more delicate transformation than memoization (Section 6), and by separating out `let`-insertion, we can have a stronger, stand-alone result regarding memoization with fewer assumptions.

### 5.1 Fixed-Point and Memoizing Fixed-Point

Since the transformations depend upon being able to syntactically identify the fixed-point operator `fix` and replace it with `mfix`, we extend  $\lambda_v^U$  with special forms `fix` ( $\lambda f \ x.e$ ) and `mfix` ( $\lambda f \ x.e$ ), both of which are taken to be in  $V^0$ . For simplicity, we will not think of these as applications of `fix` (or `mfix`) to abstractions, but rather take the  $\lambda$  as a part of the `fix` (`mfix`) syntax.

The semantics is given by the additional big-step rules in Figure 7. The rules for `fix` are standard. We do not need a rule for `mfix` at level  $> 0$  because `mfix` won’t appear in that context, as we will explain in the next subsection. This extended semantics is justified since there are well-known encodings for `fix` that satisfy it, and `mfix` can be defined in terms of `fix` (as shown in Figure 1).

The  $t$  in Figure 7 is a *table*, which is a (level-0) value that encodes a partial function  $t : V^0 \times V^0 \rightarrow V^0$ . Intuitively,  $t(F, v^0)$  returns the cached return value of the application  $F \ v^0$ . For notational convenience, we identify the term  $t$  with the partial function it

$$\begin{array}{c}
\frac{(\lambda x. [\text{fix} (\lambda f x. e^0) / f] e^0) v_1^0 \xrightarrow{1} v_2^0}{\text{fix} (\lambda f x. e^0) v_1^0 \xrightarrow{0} v_2^0} \text{ [BS-FIX]} \\
\\
\frac{e_1^n \xrightarrow{n} e_2^n}{\text{fix} (\lambda f x. e_1^n) \xrightarrow{0} \text{fix} (\lambda f x. e_2^n)} \text{ [BS-FIX+]} \\
\\
\frac{F \equiv \text{mfix} (\lambda f x. e) \quad (F, v_1^0) \notin \text{dom } t \quad (\lambda x. [F/f] e) t (\lambda s r. v_k^0 (\text{add } s (F, v_1^0) r]) r \xrightarrow{0} v_2^0}{F v_1^0 t v_k^0 \xrightarrow{0} v_2^0} \text{ [BS-MFIX}_{\notin}] \\
\\
\frac{F \equiv \text{mfix} (\lambda f x. e) \quad t(F, v_1^0) \equiv v_2^0 \quad v_k^0 t v_2^0}{F v_1^0 t v_k^0 \xrightarrow{0} v_2^0} \text{ [BS-MFIX}_{\in}]
\end{array}$$

**Figure 7.** Additional big-step rules for  $\lambda_v^U$  with fixed point.

represents. We require that we have (an encoding of) the empty mapping  $\emptyset$ . For any table  $t$  and a tuple of values  $(F, v_1^0, v_2^0)$ , we write  $t[(F, v_1^0) \mapsto v_2^0]$  for the extended table. There is a caveat here, however: the extended table does not only map  $(F, v_1^0)$  to  $v_2^0$  but maps “equivalent” function-argument pairs to  $v_2^0$  as well, according to the following definition.

**Definition 15** (Key Identification). A table lookup of the form

$$(t[(\Psi F_1, \Psi v_{a_1}^0) \mapsto \Psi v_{r_1}^0])(\Psi F_2, \Psi v_{a_2}^0)$$

returns  $v_{r_1}^0$  iff  $(F_1, v_{a_1}^0) \simeq (F_2, v_{a_2}^0)$ , where  $(\simeq)$  is some symmetric, transitive relation that implies the following conditions:

- (i)  $\text{Eval}_0(F_1 v_{a_1}^0)$  and  $\text{Eval}_0(F_2 v_{a_2}^0)$  are both defined and observationally equivalent
- (ii)  $\mathcal{M}(\text{Eval}_0(F_1 v_{a_1}^0)) \approx \mathcal{M}(\text{Eval}_0(F_2 v_{a_2}^0))$ .

This specifies the properties of key identification. If, for example, we are memoizing an even function  $F$  on the integers, it makes sense for an entry keyed by  $(F, -1)$  to be merged with the one for  $(F, 1)$ . The question then, is which entries can be merged (which keys can be equated) without corrupting the program. This mergeability criterion is  $(\simeq)$ .

Treatments of memoization in the unstaged setting appear to either gloss over this issue or require some refinement of syntactic equality [1, 12, 4]. Whereas that approach is usually adequate for unstaged programs, this is not the case with MSP. Even if a generator’s outputs for two distinct inputs are observationally equivalent so that it’s safe to merge their entries, it is very common for the outputs to be syntactically distinct when they are code values. We need a more flexible, yet well-defined, criterion. By characterizing  $(\simeq)$  axiomatically rather than fixing an implementation, we have constructed arguments that can be adapted to a variety of entry-merging strategies, including ones that use features that cannot be directly modeled by  $\lambda_v^U$  such as physical equality.

$(\simeq)$  is deliberately allowed to be non-reflexive. Effectively, entries for which reflexivity fails are never inserted into the table (or equivalently, they are inserted but always ignored upon look-up). This makes up for an inflexibility in  $\mathcal{M}$ , which memoizes every recursive function it finds. If a function  $F$  is not supposed to be memoized, then we simply assume  $(F, v_1^0) \not\simeq (F, v_2^0)$  for all  $v_1^0, v_2^0$  to model that situation.

Note that both conditions (i) and (ii) are necessary to ensure correctness. Without (i), the table implementation can choose to identify `true` and `false` in

$$\begin{array}{l}
\text{Base Types} \quad \langle \rangle, b \in B \\
\text{Types} \quad \tau \in T ::= \tau \rightarrow \tau
\end{array}$$

NB:  $\langle \rangle$  is a concrete base type, not a metavariable like  $b$ .

$$\begin{array}{c}
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ [TS-VAR]} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \text{ [TS-ABS]} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \text{ [TS-APP]} \\
\\
\frac{FV(e) \subseteq \{x \mid \Gamma(x) \in B\}}{\Gamma \vdash \langle e \rangle} \text{ [TS-BRK]} \\
\\
\frac{\Gamma \vdash \lambda f x. e : (\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_2)}{\Gamma \vdash \text{fix} (\lambda f x. e) : \tau_1 \rightarrow \tau_2} \text{ [TS-FIX]}
\end{array}$$

**Figure 8.** Minimal illustration of correctness check for  $\mathcal{M}$ .

```

let rec f =
  if b then 0
  else (f true; f false)

```

But this is illegitimate because `(f false)` diverges without memoization where it returns 0 once memoized. Without (ii), table lookup can equate all values that are observationally equivalent before transformation, which makes

```

id (fun x → id (fun y → y) x) 0

```

(where `id` is the identity function) converge if `id` is not memoized and diverge if it is.

Practically, conditions (i) and (ii) can be ensured by checking that the corresponding return values before transformation are observationally equivalent and does not contain references to memoized functions.

## 5.2 The Monadic Memoizing Transformation

The transformation is basically as follows. There are a few caveats, however, which we explain in this subsection.

**Definition 16** (CBV Monadic Memoizing Transformation). Define a partial function  $\mathcal{M} : E^0 \rightarrow E^0$  and an auxiliary transformation  $\Psi : V^0 \rightarrow V^0$  on values by

$$\begin{array}{l}
\mathcal{M}x \stackrel{\text{def}}{=} \text{ret } x \\
\mathcal{M}v^0 \stackrel{\text{def}}{=} \text{ret } (\Psi v^0) \\
\mathcal{M}(e_1 e_2) \stackrel{\text{def}}{=} \text{bind } (\mathcal{M}e_1) (\text{bind } (\mathcal{M}e_2)) \\
\mathcal{M}\langle e \rangle \stackrel{\text{def}}{=} \text{ret } \langle e \rangle \\
\Psi(\text{fix} (\lambda f x. e)) \stackrel{\text{def}}{=} \text{mfix} (\lambda f s. \mathcal{M}e) \\
\Psi(\lambda x. e) \stackrel{\text{def}}{=} \lambda x. \mathcal{M}e \\
\Psi\langle e^0 \rangle \stackrel{\text{def}}{=} \langle e^0 \rangle
\end{array}$$

The idea is to replace all applications with `bind` and to wrap all values and variables in `ret`. The `bind` and `ret` are the standard monad operations, shown in Figure 2.

Unlike the discussions in Section 4, where we did not need to rely on type information, the correctness of  $\mathcal{M}$  requires some rudimentary type analysis. Figure 8 shows a simple, minimal type system that illustrates the kinds of checks that need to be performed before applying  $\mathcal{M}$ .

The type system is kept as small as possible, both to simplify the presentation and to avoid committing to the details of a particular type system. It is basically the simply typed lambda calculus with a multi-stage extension. It is not necessary to check the details of the code inside brackets, so we collapse all code types into the catch-all code type  $\langle \cdot \rangle$ .

The main driving force behind the type check is that we don't want the transformation to touch the contents of  $\langle \cdot \rangle$ . That part of the code is in the future stage, where we want to have efficient code, and leaving *ret* and *bind* in there could result in a performance disaster (although perhaps not as catastrophic as code duplication). This constraint raises some subtle issues that do not arise when monadifying an unstaged program.

For example, consider the code

```
let f = fun x → x in
let g = .! .<fun x → x>. in
let y = g .<f 0 + 1>. in
...
```

Blindly monadifying this code with the above constraint makes it ill-typed.

```
bind (ret (fun x → ret x)) (fun f →
bind (ret (.! .<fun x → x>.) (fun g →
bind (g .<f 0 + 1>.) (fun y →
...)))
```

The application of *f* is problematic because it returns a monad to a context expecting an integer. If unconstrained, CSP can bring monadified values into unmonadified contexts. For  $\mathcal{M}$  to be sound, CSP is only permissible on values  $v^0$  s.t.  $\Psi v^0 \equiv v^0$ . We ensure this by checking that CSP variables are always of a non-functional type. Our minimal type system ensures this by checking the absence of function variables in the premise of TS-BRK.

The call to *g* is also illegitimate: it returns a non-monadic value when *bind* expects a monad for its first argument. Effectively, *.!* is providing a loophole for passing unmonadified values to monadified contexts, which is the dual problem of CSP. In general, whether to transform code under *run* is a complex decision that cannot be made without intricate knowledge of the specific program. We require that the transformed part of the program not contain stage-0 *run*—hence the lack of rules for *run e* in Definition 16 and the type system. This is a reasonable assumption to make for generators, whose job is to create code and not to run it.

Hence, it is necessary (and sufficient, as we will prove) to have a type system with the following features in order to implement the safety check.

- It must distinguish function types from base/code types, and point out which variable has which.
- It must detect (and reject) *run* and  $\sim$  outside brackets.

On the other hand, it is not necessary at all to be able to check what goes on inside brackets. The type system in Figure 8 satisfies these requirements and also implements the check. It should be clear, though, that although our type system has a specialized rule for building in the check for functional CSP and a lack of rule in order to reject *run*, any sensible type system (like MetaOCaml's) should be able to detect a monadic value usage in a non-monadic context (or vice versa) as a type mismatch, without special instruction from the programmer.

The safety property that these checks buy us can be summarized as follows.

**Lemma 17** (Substitution Commutes with  $\mathcal{M}$ ). *If  $\Gamma \vdash v^0 : \tau$  and  $\Gamma, x : \tau \vdash e : \tau'$ , then  $[\Psi v^0/x](\mathcal{M}e) \equiv \mathcal{M}([v^0/x]e)$ .*

*Proof.* Mostly straightforward induction the judgment  $\Gamma, x : \tau \vdash e : \tau'$ . The only interesting case is when  $e \equiv \langle e' \rangle$ . By inversion,  $\Gamma \vdash e' : \tau''$  where  $\tau' = \langle \tau'' \rangle$ . If  $x \in FV(e')$ , then  $\tau \in B$ , hence  $v^0 \equiv \langle e^0 \rangle$ , therefore  $\Psi v^0 \equiv v^0$ . Hence

$$\begin{aligned} \mathcal{M}([v^0/x]e) &\equiv \mathcal{M}(\langle [v^0/x]e' \rangle) \\ &\equiv \text{ret } \langle [v^0/x]e' \rangle \equiv [v^0/x](\text{ret } \langle e' \rangle) \equiv [\Psi v^0/x](\mathcal{M}e). \end{aligned}$$

□

Of course, this property is useless if it is disrupted by reduction. (The progress lemma does not hold—we do not need it.)

**Proposition 18** (Subject Reduction). *If  $\Gamma \vdash e : \tau$  and  $e \longrightarrow e'$ , then  $\Gamma \vdash e' : \tau$ .*

*Proof.* Straightforward induction on the reduction judgment. Unlike with more complicated type systems, technical lemmas like promotion and demotion do not arise since *run* and  $\sim$  are not typable (at level 0). □

### 5.3 Correctness

The type system captures a sufficient condition for  $\mathcal{M}$  to preserve semantics.

**Theorem 19.** *Suppose  $\Gamma \vdash e^0 : \tau$ . Then  $e^0$  terminates iff  $\mathcal{M}e^0 \emptyset$  kвал does. Furthermore, if  $e^0 \xrightarrow{0} v_1^0$  then  $\mathcal{M}e \emptyset$  kвал  $\approx \Psi v_2^0$  for some  $v_2^0 \approx v_1^0$ .*

The significance of this theorem is that a generator returning  $\langle e \rangle$ , when memoized, will return  $\Psi \langle e' \rangle$  for some  $e'$  such that  $\Psi \langle e' \rangle \equiv \langle e' \rangle \approx \langle e \rangle$ . Thus memoization preserves the semantics of a generator.

Here is a proof sketch. We refer the reader to an accompanying technical report [13] for the full details. The big-step semantics tends to be a better match for stateful computations so we will use that instead of the equational theory.

We first face the same obstacle that Plotkin did in his proof of correctness for CPS transformation [23], which is that superfluous redexes makes the reduction of  $\mathcal{M}e$  correspond poorly to reduction of  $e$ . Mimicking his approach, we define optimized monadic transformations  $\mathcal{M}'$  and  $\Psi'$  as

$$\mathcal{M}'x \stackrel{\text{def}}{=} \text{ret } x$$

$$\mathcal{M}'v^0 \stackrel{\text{def}}{=} \lambda s k.k s (\Psi'v)$$

$$\mathcal{M}'\langle e^1 \rangle \stackrel{\text{def}}{=} \text{ret } \langle e^1 \rangle \tag{*0}$$

$$\mathcal{M}'(e_1^0 e_2^0) \stackrel{\text{def}}{=} \text{bind } (\mathcal{M}'e_1^0) (\text{bind } (\mathcal{M}'e_2^0)) \tag{*1}$$

$$\mathcal{M}'(e_1^0 e_2^0) \stackrel{\text{def}}{=} B(\mathcal{M}'e_1^0) (\text{bind } (\mathcal{M}'e_2^0)) \tag{*2}$$

$$\mathcal{M}'(e_1^0 e_2^0) \stackrel{\text{def}}{=} \text{bind } (\mathcal{M}'e_1^0) (B(\mathcal{M}'e_2^0)) \tag{*3}$$

$$\mathcal{M}'(e_1^0 e_2^0) \stackrel{\text{def}}{=} \lambda s k. (\mathcal{M}'e_1^0) s (\lambda s' r. (B(\mathcal{M}'e_2^0)) r s' k) \tag{*4}$$

$$\Psi'(\text{fix } (\lambda f x.e)) \stackrel{\text{def}}{=} \text{mfix } (\lambda f s.\mathcal{M}'e)$$

$$\Psi'(\lambda x.e) \stackrel{\text{def}}{=} \lambda x.\mathcal{M}'e$$

$$\Psi'\langle e^0 \rangle \stackrel{\text{def}}{=} \langle e^0 \rangle$$

where  $B(e) \stackrel{\text{def}}{=} (\lambda f s k.e s (\lambda s' r.f r s' k))$ . (Note that  $B$  is a macro, and the two sides of this equation are *syntactically* equal.) The rule (\*0) is applied when  $e^1 \notin E^0$ . The rules (\*1)–(\*4) are chosen according to the following table.

	$\mathcal{M}'e_1^0 \in V^0$	$\mathcal{M}'e_1^0 \notin V^0$
$\mathcal{M}'e_2^0 \in V^0$	(*4)	(*3)
$\mathcal{M}'e_1^0 \notin V^0$	(*2)	(*1)

Then it is easy to see that  $\mathcal{M}e = \mathcal{M}'e$ , so we can prove Theorem 19 by proving it for  $\mathcal{M}'e$  instead.

We modify the definition of memo table to use  $\mathcal{M}'/\Psi'$  instead of  $\mathcal{M}/\Psi$ . We define a stateful big-step relation  $t; e \Downarrow t'; e'$  to reason more directly about a monadic program (Figure 9). This monadic

$$\begin{array}{c}
\frac{v^0 \equiv (\lambda x.e^0) \text{ or } (\text{fix } (\lambda f x.e^0))}{t; v^0 \Downarrow t; v^0} \quad \frac{\langle e \rangle \xrightarrow{0} \langle e' \rangle}{t; \langle e \rangle \Downarrow t; \langle e' \rangle} \\
\frac{t_0; e_1^0 \Downarrow t_1; \lambda x.e_3^0 \quad t_1; e_2^0 \Downarrow t_2; v_2^0 \quad t_2; [v_2^0/e_3^0] \Downarrow t_3; v^0}{t_0; e_1 e_2 \Downarrow t_3; v^0} \\
\frac{t_0; e_1^0 \Downarrow t_1; \text{fix } (\lambda f x.e_3^0) \quad t_1; e_2^0 \Downarrow t_2; v_2^0 \quad t_2; (\Psi'(\text{fix } (\lambda f x.e_b^0)), \Psi'v_2^0) \notin \text{dom } t_2 \quad t_2; (\lambda x. [\text{fix } (\lambda f x.e_3^0)]e_3^0) v_2^0 \Downarrow t_3; v^0}{t_0; e_1 e_2 \Downarrow t_3; v^0} \\
\frac{t_0; e_1^0 \Downarrow t_1; \text{fix } (\lambda f x.e_3^0) \quad t_1; e_2^0 \Downarrow t_2; v_2^0 \quad t_2 (\Psi'(\text{fix } (\lambda f x.e_b^0)), \Psi'v_2^0) \equiv v^0}{t_0; e_1 e_2 \Downarrow t_2; v^0}
\end{array}$$

**Figure 9.** Big-step monadic semantics.

semantics can be shown to be a partial function  $(\Downarrow) : \text{Table} \times E^0 \rightarrow \text{Table} \times V^0$  that is sound and complete.

**Lemma 20.** *Suppose  $\Gamma \vdash t_1^0 : \tau$ . Then  $t; e_1^0 \Downarrow t'; e_2^0 \wedge v_k^0 t' (\Psi'e_2^0) \xrightarrow{0} v^0 \iff \mathcal{M}'e t v_k^0 \xrightarrow{0} v^0$ .*

*Proof.* The  $(\implies)$  direction is by induction on the judgment  $t; e_1^0 \Downarrow t'; e_2^0$ . The converse is likewise by induction on  $\mathcal{M}'e t v_k^0 \xrightarrow{0} v^0$ . Both directions require Proposition 17 to reason about substitution.  $\square$

Then we define a good table to be one that does not contain entries that cache incorrect values. We give one definition that only respects Definition 15(i), and show that this property is preserved by  $(-; - \Downarrow -; -)$ . This is sufficient to prove soundness of the transformation. We then give another definition that only observes Definition 15(ii) and use it similarly to prove completeness.

**Lemma 21.** *Define  $G(t) \stackrel{\text{def}}{\iff} \forall F, v_1^0, v_2^0 \in V^0. t(\Psi'F, \Psi'v_1^0) \equiv \Psi'v_2^0 \implies F v_1^0 \approx v_2^0$ . Then  $G(t) \wedge t; e^0 \Downarrow t'; v^0 \wedge \Gamma \vdash e^0 : \tau \implies e^0 \approx v^0 \wedge G(t')$ .*

*Proof.* Induction on the judgment  $t; e^0 \Downarrow t'; v^0$ .  $\square$

**Lemma 22.** *Let  $G'(t) \stackrel{\text{def}}{\iff} (\forall F, v_1^0, v_2^0 \in V^0. t(\Psi'F, \Psi'v_1^0) \equiv \Psi'v_2^0 \implies \Psi'(E\text{val}_0(F v_1^0)) \approx \Psi'v_2^0)$ . Then  $G'(t) \wedge e^0 \xrightarrow{0} v_1^0 \wedge \Gamma \vdash e^0 : \tau$  implies  $\exists v_2^0. t; e^0 \Downarrow t'; v_2^0 \wedge G'(t') \wedge \Psi'v_2^0 \approx \Psi'v_1^0$ .*

*Proof.* Induction on the judgment  $e^0 \xrightarrow{0} v^0$ . We need to prove Lemma 23 along the way.  $\square$

**Lemma 23.** *If  $\mathcal{M}'e_1^0 \approx \mathcal{M}'e_2^0$ ,  $t; e_1^0 \Downarrow t_1; v_1^0$ , and  $t; e_1^0 \Downarrow t_2; v_2^0$ , then  $\mathcal{M}'v_1^0 \approx \mathcal{M}'v_2^0$  and  $t_1 \approx t_2$ . Furthermore, whenever  $t_1(\Psi'F, \Psi'v_a^0) \equiv v_a^0$  we have  $t_2(\Psi'F)$ .*

*Proof of Theorem 19.*  $G(\emptyset)$  and  $G'(\emptyset)$  so the theorem follows from Lemmas 21 and 22.  $\square$

## 5.4 Example: Verifying Memoized-Staged Gibonacci

The `gibgenm` generator in Figure 1 is the transform of `gibgen`, apart from these differences, it uses `mfIx1` instead of `mfIx`, it memoizes a ternary function rather than a unary function, and it uses a special rule to handle conditionals. The first discrepancy is discussed in the next section. The other differences are minor and can be readily justified. We illustrate the use of Theorem 19 to verify this generator.

**Proposition 24.** *For any non-negative integer  $n$ ,*

$$\text{gibgen } n \text{ .<x>. .<y>.} \approx \text{gibgenm } n \text{ .<x>. .<y>.$$

*Proof.* By Corollary 14, it suffices to prove equivalence with `gibgen`, which in turn by Theorem 19 reduces to a type check for `gibgenm`. As a rule of thumb, an expression is typable if it is bracket-closed, no `!` appears and no “function” is used in CSP. The first two are simple to check. For the last one, we observe that the type of `+` does not matter because it only appears inside brackets, so we can assign it an opaque “base” type. Therefore, `gibgenm` is typable under  $\Gamma = + : b, 0 : \text{int}, 1 : \text{in}$ , which concludes the proof.  $\square$

This argument may seem to break down if `+` appears both outside and inside brackets, but it is easily adapted to that case. We temporarily add the binding `let plus = (+)` to the code we are type checking, and modify the code to use `plus` outside brackets while keeping references to `+` inside brackets. When all is done, we use  $\beta_v$  equivalence to eliminate the superfluous binding. In general, CSP of functional values is problematic only if the value is bound within the region that is subject to transformation.

With Theorem 19, we get the divergence of `gibgenm` for negative  $n$  for free provided it is proved for `gibgen`.

## 6. Let-insertion

$\mathcal{M}$  does not eliminate code duplication itself, but is rather a tool for having the right setup for it. Code duplication is finally eliminated when we replace `mfIx` with `mfIx1`. This `let`-insertion step turns out to be the most problematic step of staged memoization.

As mentioned earlier, the difference between these combinators boils down to

```
.<let z = .~c in
  .~(k (add (n, x, y, .<z>) t') .<z>)>.
```

v.s.

```
k (add (n, x, y, c)) c
```

where `c` is bound to the code value returned by the memoized function. There is a deep-down problem with the code motion performed here, which shows up when `c` contains a side effect. Because we are using a purely functional language, a side effect is synonymous with divergence. For example,

```
let rec loop () = loop () in
let foo f b = if b then .<loop ()>.
               else .<fun _ -> .~(f true)>.>.
in
  (* execution *)
  mfIx foo false => .<fun _ -> loop()>. (* A *)
  mfIx1 foo false => .<let z = loop () (* B *)
                    in fun _ -> z>.
```

Code A is convergent while code B is divergent. One can construct a similar example using a conditional in place of `fun`. A side effect of `let`-insertion is to invert the order in which expressions appear in the generated code, which can cause an effectful (divergent) expression to be moved out of a delaying construct and be executed

earlier than it’s supposed to (if it should be run at all). An alternative scenario is that the code value would actually be discarded without `let`-insertion and hence does not appear in the final generated code, but `fix1` has no way of knowing this and inserts the `let` binding anyway. We call this the effect motion problem.

This is an old problem that had been found in the context of partial evaluation by Bondorf [2], but it seems to have been overlooked in MSP. Bondorf handles effect motion by cutting off the continuation-passing at points where the generator crosses over a stage-1 `fun` or `if`, and inserting all the `let` that had been accumulated up to that point. He identifies these cross-over points by defining “safe” contexts  $S$ , where  $S[e]$  is guaranteed to be strict in  $e$ . Given a safe context  $S$ , he can safely splice any expression in a context of the form  $\langle S[] \rangle$ . In that case, he splices in a variable, like  $\langle S[x] \rangle$ , and saves the binding  $x : e$  to be turned into a `let` binding later. Otherwise, if given an unsafe context, he immediately splices  $e$  into the context, along with any pending `let` bindings.

Generally speaking, we cannot take Bondorf’s approach in monadic staging because MSP forbids intentional analysis of code values. There is no general way to identify safe and unsafe contexts other than to ask the programmer, but that would require copious annotations of some sort (and raises the question of whether those annotations are correct). This destroys the simplicity of monadic staging. Developing a lightweight extension to monadic staging with this approach that still preserves the elegance of monadic staging is beyond the scope of this paper, and we leave it for future work.

In the case of `gibgenm`, however, we can apply Bondorf’s solution. In the unmonadic generator, the code value returned from a recursive call is always spliced into a safe context, either  $\langle [] + e \rangle$ . or  $\langle e + [] \rangle$ . Observing that if  $S$  is a strict context then  $S[e] \approx (\lambda x. S[x]) e$ , and we can prove the monadic, memoized, `let`-inserted generator correct.

An alternative is to do termination analysis on the code fragments that are being moved around (which includes every intermediate fragment generated by a recursive call). A practical way to achieve this is to require the stage-1 code to be typable in a strongly normalizing type system, like the simply typed lambda calculus. In MetaOCaml, this can be done by checking that no `let rec` and partial library functions appear inside brackets. CBV and CBN coincide in strongly normalizing languages, so the `let`-insertion is justified via CBN  $\beta$ -reduction.

We can expect both of these lines of argument to work for relatively simple programs like DSP circuits or dynamic programming algorithms. However, they do not work for more feature-hungry programs like interpreters, and for those programs, monadic staging is generally not semantics-preserving. Whether this limitation can be lifted is left for future investigation.

## 7. Related Work

The current work is in large part a follow-up to Swadi et al. [24], who first described monadic staging. They were the first to propose in the MSP literature to monadically encapsulate and systematize the memoization-CPS transformation.

The theoretical framework used here draws heavily from the work by Taha [25], who first laid down an equational MSP theory in satisfactory form. The key ideas were to restrict reductions to level-0 expressions and to sacrifice intentional analysis. Intentional analysis may be recovered to a large extent, however, by adding wrappers as illustrated in [24]. It will be interesting to see if this can be exploited to mimic Bondorf’s solution to the effect motion problem.  $\lambda^U$  is untyped, purely functional, and CBN, and the subtlety regarding whether a variable can be  $\beta$ -substituted does not arise.

MSP originally grew out of partial evaluation as a mechanism for explaining and implementing partial evaluation strategies [22]. There is a long history of pursuit for type systems for MSP [9, 21,

28, 33]. Of these, the  $\lambda^\alpha$  system by Taha and Nielsen [28] forms the basis of MetaOCaml. Type-safe extensions with imperative features are still largely in the works, with Kameyama et al. [15] being a notable recent example of progress.

The memoization-CPS transformation idea itself was known to the partial evaluation and MSP communities much earlier. CPS-translation of a generator to improve `let`-insertion was first explored by Bondorf [2]. His partial evaluator also memoized generated functions, but not the generator. This technique has since become standard in the partial evaluation literature. Bondorf was aware of the effect-motion issue, and introduced some rules for avoiding them, as explained in Section 6. Yet, when his ideas were first formalized in the MSP setting by Swadi et al. [24], the effect motion issue was not given attention. Swadi et al. were still able to achieve substantial results without running into this problem.

Earlier efforts to systematize memoization include Acar et al. [1] and Frost [12]. The latter seems to be the first to present the idea of using monads. Frost states in [12] that this idea was in turn given to him by an anonymous reviewer of an earlier paper. This development was unknown to the authors of [24], and the idea was re-discovered independently. Subsequently, this technique was refined and generalized as memoizing mix-ins by Brown and Cook [4]. Acar et al. [1] focused on specifying efficient key identification (entry merging), which we modeled as  $\simeq$ . We deliberately left this aspect underspecified, in order to capture a broad range of memoization schemes, including the one by Acar et al.

Moggi [20] first introduced monads to computer science as a means to structure denotational semantics. It has since been adopted as a prominent functional programming idiom, especially in the Haskell community through the works of Wadler and Peyton Jones [31, 14, 32].

## 8. Conclusion and Future Work

We have shown how several results from the metatheory of multi-stage languages can help us justify the correctness of a staged program. In particular, we were able to show that a staged program is equivalent to its erasure if it can be proved equal to an unstaged form, which is a direct consequence of confluence of the rewrite system underlying MSP [25]. Furthermore, we proved, specifically for monadic staging, that a simple type check is enough to guarantee the correctness of the monadic transformation phase. Finally, we also found that justifying the `let`-insertion component requires restricting our attention to a particular set of transformed programs. This concern becomes the responsibility of the programmer applying the method. With these results, we have laid down a foundation that should give programmers a jump-start on verifying staged programs.

The largest, non-trivial obligation left to the programmers is justifying `let`-insertion. This can require intimate knowledge of MSP metatheory or making program-specific adjustments to the method. It is certainly desirable to have a strengthened method that either avoids effect motion completely or has a criterion for avoiding it. An extension of the monadic staging method presented here with more sophisticated and complete set of combinators exist, proposed by Carette and Kiselyov [7], but they did not address effect motion either. It is of interest to see if their combinators may be refined to deal to effect motion.

## Acknowledgments

Earlier presentations of Proposition 12 and Corollary 13 were greatly improved by suggestions by Gregory Malecha. Mathias Ricken has provided us with important feedback on the same topic. We thank Yukiyoshi Kameyama, Eugenio Moggi, and Ronald Garcia for giving us valuable input. We thank Ray Hardesty for improving our writing in this paper.

## References

- [1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–25, New York, NY, USA, 2003. ACM.
- [2] Anders Bondorf. Improving binding times without explicit CPS-conversion. In *LFP '92: Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 1–10, New York, NY, USA, 1992. ACM.
- [3] Edwin Brady and Kevin Hammond. A verified staged interpreter is a verified compiler. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 111–120, New York, NY, USA, 2006. ACM.
- [4] Daniel Brown and William R. Cook. Monadic memoization mixins. Regular report TR-07-11, The University of Texas at Austin, Department of Computer Sciences, February 2007.
- [5] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. ML-like inference for classifiers. In David Schmidt, editor, *ESOP '04: Proceedings of the 13th European Symposium on Programming*, volume 2986/2004 of *Lecture Notes in Computer Science*, pages 79–93. Springer, 2004.
- [6] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In Krzysztof Czarnecki, Frank Pfenning, , and Yannis Smaragdakis, editors, *GPCE '03: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, pages 57–76, New York, NY, USA, 2003. Springer-Verlag New York, Inc.
- [7] Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In Robert Glück and Michael R. Lowry, editors, *GPCE '05: Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, volume 3676, pages 256–274. Springer, 2005.
- [8] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated tagless staged interpreters for simpler typed languages. In Zhong Shao, editor, *APLAS '07: Proceedings of the 5th ASIAN Symposium on Programming Languages and Systems*, volume 4807/2007 of *Lecture Notes in Computer Science*, pages 222–238. Springer-Verlag, 2007.
- [9] Rowan Davies. A temporal-logic approach to binding-time analysis. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, Washington, DC, USA, 1996. IEEE Computer Society.
- [10] R. Kent Dybvig. Writing hygienic macros in scheme with syntax-case. Technical Report TR356, Indiana University Computer Science Department, 1992.
- [11] Hyunjun Eo, Ik soon Kim, and Kwangkeun Yi. Type and effect system for multi-staged exceptions. In *APLAS '06: 4th Asian Symposium on Programming Languages and Systems*, volume 4297 of *Lecture Notes in Computer Science*, pages 61–78. Springer-Verlag, 2006.
- [12] Richard A. Frost. Monadic memoization towards correctness-preserving reduction of search. In Yang Xiang and Brahim Chaib-draa, editors, *Proceedings of the 16th Conference of the Canadian Society for Computational Studies of Intelligence*, volume 2671 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2003.
- [13] Jun Inoue and Walid Taha. Reasoning about staged programs. Technical Report TR09-3, Rice University Computer Science Department, July 2009.
- [14] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 71–84, New York, NY, USA, 1993. ACM.
- [15] Yukiyooshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage: staging with delimited control. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 111–120, New York, NY, USA, 2008. ACM.
- [16] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 257–268, New York, NY, USA, 2006. ACM.
- [17] Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *EMSOFT '04: Proceedings of the 4th ACM International Conference on Embedded Software*, pages 249–258, New York, NY, USA, 2004. ACM.
- [18] Oleg Kiselyov and Walid Taha. Relating fftw and split-radix. In *ICSS '04: Proceedings of the International Conference on Embedded Software and Systems*, volume 3605 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
- [19] Xavier Leroy et al. Objective caml, 2000.
- [20] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.
- [21] Aleksandar Nanevski. Meta-programming with names and necessity. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 206–217, New York, NY, USA, 2002. ACM.
- [22] Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*. Cambridge University Press, New York, NY, USA, 1992.
- [23] Gordon D. Plotkin. Call-by-name, call-by-value and the  $\lambda$ -calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.
- [24] Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pašalić. A monadic approach for avoiding code duplication when staging memoized functions. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 160–169, New York, NY, USA, 2006. ACM.
- [25] Walid Taha. *Multistage programming: Its theory and applications*. PhD thesis, Oregon Graduate Institute, 1999. Supervisor-Tim Sheard.
- [26] Walid Taha et al. MetaOCaml: A compiled, type-safe multi-stage programming language, 2004.
- [27] Walid Taha and Patricia Johann. Staged notational definitions. In *GPCE '03: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, pages 97–116, New York, NY, USA, 2003. Springer-Verlag New York, Inc.
- [28] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 26–37, New York, NY, USA, 2003. ACM.
- [29] Masako Takahashi. Parallel reductions in lambda-calculus. *Information and Computation*, 118(1):120–127, 1995.
- [30] Takeshi Tsukada and Atsushi Igarashi. A logical foundation for environment classifiers. In Pierre-Louis Curien, editor, *Proceedings of the 9th International Conference on Typed Lambda-Calculi and Applications (TLCA '09)*, volume 5608 of *Lecture Notes in Computer Science*, pages 341–355. Springer-Verlag, July 2009. To appear.
- [31] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, New York, NY, USA, 1992. ACM.
- [32] Philip Wadler. Monads for functional programming. In *First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.
- [33] Yoshihiro Yuse and Atsushi Igarashi. A modal type system for multi-level generating extensions with persistent code. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 201–212, New York, NY, USA, 2006. ACM.