

Computing with Capsules

Jean-Baptiste Jeannin
Department of Computer Science
Cornell University
Ithaca, New York 14853–7501, USA
Email: jeannin@cs.cornell.edu

Dexter Kozen
Department of Computer Science
Cornell University
Ithaca, New York 14853–7501, USA
Email: kozen@cs.cornell.edu

January 26, 2011

Abstract

Capsules provide a clean algebraic representation of the state of a computation in higher-order functional and imperative languages. They play the same role as closures or heap- or stack-allocated environments but are much simpler. A capsule is essentially a finite coalgebraic representation of a regular closed λ -coterms. One can give an operational semantics based on capsules for a higher-order programming language with functional and imperative features, including mutable bindings. Lexical scoping is captured purely algebraically without stacks, heaps, or closures. All operations of interest are typable with simple types, yet the language is Turing complete. Recursive functions are represented directly as capsules without the need for unnatural and untypable fixpoint combinators.

1 Introduction

This paper introduces *capsules*, an algebraic representation of the state of a computation in higher-order functional and imperative programming languages. Representations of state have been studied in the past by many authors (e.g. [1–12]). However, unlike previous approaches, capsules are purely algebraic. They correctly capture lexical scoping without closures and without any combinatorial constructs such as stacks or heaps. Rigorous formal reasoning is possible with algebraic and coalgebraic methods.

Formally, a capsule is a particular syntactic representation of a finite coalgebra of the same signature as the λ -calculus. A capsule represents a regular closed λ -coterms (infinite λ -term) under the unique morphism to the final coalgebra of this signature. This final coalgebra has been studied under the name *infinitary λ -calculus*, focusing mostly on infinitary rewriting [13, 14]. It has been observed that the infinitary version does not share many of the desirable properties of its finitary cousin; for example, it is not confluent, and there exist coterms with no computational significance (Fig. 1). However, all coterms represented by capsules are computationally meaningful.

One can give an operational semantics based on capsules for a higher-order programming language with functional and imperative features, including recursion and mutable variables. All

operations of interest are typable with simple types, yet the language is Turing complete. Recursive functions can be represented directly without the need for fixpoint combinators. Fixpoint combinators are rather unnatural because they involve self-application and are therefore untypable with simple types. Moreover, the traditional Y combinator forces a normal-order (lazy) evaluation strategy to ensure termination. Other more complicated fixpoint combinators can be used with applicative order by encapsulating the self-application in a thunk to delay evaluation, but this is even more unnatural. In contrast, the representation of recursive functions with capsules is direct, simply typable, and corresponds more closely to actual implementations. Turing completeness is impossible with finite types and finite terms, as the simply-typed λ -calculus is strongly normalizing; so we must have either infinitary types or infinitary terms. Whereas the former is more conventional, we believe the latter is more natural.

Dynamic scoping, which was the preferred scoping discipline in early versions of LISP and Python and which still exists in many languages today, can be regarded as a flawed implementation of lazy β -reduction that fails to observe the principle of safe substitution (α -conversion to avoid capture of free variables). We explain this view more fully with a detailed example in §3. In contrast, lexical scoping correctly models β -reduction with safe substitution in the λ -calculus. Both capsules and closures provide lexical scoping, but capsules do it in a much simpler way. The formal definition of closures involves a mutual coinductive definition of environments and values. Moreover, capsules work correctly in the presence of mutable variables, whereas closures, naively implemented, do not (a counterexample is given in §4.4). Perhaps this is one reason that mutable variables, which existed in LISP and Scheme in the form of `set!`, were dropped in the ML family. To correctly handle mutable variables, closures require some form of indirection, and care must be taken to perform updates nondestructively.¹ These usually require combinatorial data structures and some kind of auxiliary storage management mechanism, introducing further complications in the model. In contrast, no such constructs are necessary with capsules.

Another insight is a compelling answer to the question: What do we mean by *correctness* in the handling of mutable variables? This is subject to definition, and one might take current implementations of functional languages as guidance; but our definition is mathematical and conservatively extends the scoping rules of the λ -calculus.

Most interestingly, capsules provide a strong link between functional and imperative programming. Valuations of mutable variables used in the semantics of imperative programs are similar to closures used to effect lexical scoping in functional programs. Capsules exploit this similarity and provide a common model for both. We also get a clean definition of garbage collection: there is a natural notion of morphism, and the garbage-collected version of a capsule is the unique (up to isomorphism) initial object among its monomorphic preimages.

In this paper, by *state* we generally mean *state of a computation*, as opposed to the state of dynamically mutable data objects. The state of mutable data objects in a computation is certainly part of the state of the computation, but in our treatment, data objects themselves are generally immutable (although there is no intrinsic reason that they need to be). It is easy to confuse the two notions. There is a long tradition of using a common representation of programs and data, e.g. *S*-expressions in LISP. One often models objects as collections of mutable bindings, as for example in the ζ -calculus of Abadi and Cardelli [16], the same mechanism we use in capsules. However,

¹Wikipedia [15] identifies this as an issue with the implementation of dynamic scoping, but the issue also arises with static scoping.

by keeping data immutable, we are able to provide a purer, more algebraic notion of computation, one that does not depend on mutable data for its implementation.

There is much previous work on reasoning about references and local state; see [1–4, 17–21]. State is typically modeled by some form of heap from which storage locations can be allocated and deallocated [1–3, 7–10]. Others have used game semantics to reason about local state [22–24]. Moggi [6] proposed monads, which can be used to model state and are implemented in Haskell. Our approach is most closely related to the work of Mason and Talcott [1–3] and Felleisen and Hieb [4]. These approaches aspire to the same goals, but we hope to convince the reader that ours is considerably simpler.

This paper is organized as follows. In §2, we give formal definitions of capsules and the λ -coalgebras and λ -coterms they represent. In §3 we give a detailed motivating example. In §4 we prove two theorems. The first (Theorem 4.1) establishes that capsule evaluation faithfully models β -reduction in the λ -calculus with safe substitution. The second (Theorem 4.7) defines closure conversion for capsules and proves soundness of the translation, provided there is no variable assignment (as previously mentioned, capsules work correctly in the presence of mutable bindings, closures do not). The proof techniques in this section are purely algebraic and involve some interesting applications of coinduction. Finally, in §5, we describe a simply-typed functional/imperative language with mutable bindings and give an operational semantics in terms of capsules.

2 Definitions

2.1 Capsules

Consider the simply-typed λ -calculus with typed constants (e.g., $3 : \text{int}$, $\text{true} : \text{bool}$, $+$: $\text{int} \rightarrow \text{int} \rightarrow \text{int}$, \leq : $\text{int} \rightarrow \text{int} \rightarrow \text{bool}$). The set of λ -abstractions is denoted $\lambda\text{-Abs}$ and the set of constants is denoted Const . A λ -term is *irreducible* if it is either a λ -abstraction $\lambda x.e$ or a constant c . The set of irreducible terms is $\text{Irred} = \lambda\text{-Abs} + \text{Const}$. Note that variables x are not irreducible.

Let $\text{FV}(e)$ denote the set of free variables of e . A *capsule* is a pair $\langle e, \sigma \rangle$, where e is a λ -term and $\sigma : \text{Var} \rightarrow \text{Irred}$ is a partial function with finite domain $\text{dom } \sigma$, such that

- (i) $\text{FV}(e) \subseteq \text{dom } \sigma$
- (ii) if $x \in \text{dom } \sigma$, then $\text{FV}(\sigma(x)) \subseteq \text{dom } \sigma$.

A capsule $\langle e, \sigma \rangle$ is *irreducible* if e is.

Note that cycles are allowed; this is how recursive functions are represented. For example, we might have $\sigma(f) = \lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n - 1)$.

More generally, a *precapsule* is a pair $\langle e, \sigma \rangle$, where e is a λ -term and $\sigma : \text{Var} \rightarrow \text{Irred}$ is a partial function. A *precapsule* is *closed* if it satisfies (i) and (ii) above, and *finite* if the domain of σ is finite. A capsule is thus a closed finite precapsule.

2.2 Σ -Coterms and Σ -Coalgebras

To describe the coalgebraic nature of capsules, we need to define λ -coalgebras and λ -coterms. We define Σ -coalgebras and Σ -coterms for an arbitrary signature Σ ; we will be interested in the special case of the signature of the λ -calculus,² although for a different signature the same formalism can be used to describe recursive types.

A *signature* Σ is a set equipped with an arity function $\text{arity} : \Sigma \rightarrow \omega$. *Terms* are defined inductively as usual. The set of Σ -terms is denoted T_Σ . Equipped with their syntactic interpretations (e.g. $f^{T_\Sigma}(t_1, \dots, t_n) = f(t_1, \dots, t_n)$ for $n = \text{arity}(f)$), the terms T_Σ form the *free Σ -algebra* in the sense that for any Σ -algebra A , there is a unique Σ -algebra homomorphism $T_\Sigma \rightarrow A$.

A Σ -*coterm* is a partial map $t : \omega^* \rightarrow \Sigma$ with domain $\text{dom } t$ such that

- $\text{dom } t$ is nonempty and prefix-closed,
- if $\alpha \in \text{dom } t$, then $\alpha i \in \text{dom } t$ iff $i < \text{arity}(t(\alpha))$.

Informally, we can view t as a labeled tree with root ε and edges $(\alpha, \alpha i)$, where the map t labels the nodes of the tree. The family of Σ -coterms is denoted C_Σ .

If $\alpha \in \text{dom } t$, the *subterm of t rooted at α* is the coterm $\beta \mapsto t(\alpha\beta)$ and is denoted $t \upharpoonright \alpha$; thus $(t \upharpoonright \alpha)(\beta) = t(\alpha\beta)$.

The Σ -coterms form the final coalgebra in the category of Σ -coalgebras. A Σ -*coalgebra* is a tuple (S, δ, ℓ) , where S is a set of *states* (not necessarily finite), $\delta : S \times \omega \rightarrow S$ is a partial *transition function*, and $\ell : S \rightarrow \Sigma$ is a *labeling function* such that

- S is nonempty,
- $(s, i) \in \text{dom } \delta$ iff $i < \text{arity}(\ell(s))$.

We can extend δ inductively to a partial function $\widehat{\delta} : S \times \omega^* \rightarrow S$:

$$\widehat{\delta}(s, \varepsilon) = s \qquad \widehat{\delta}(s, \alpha i) = \delta(\widehat{\delta}(s, \alpha), i).$$

The Σ -coterms form a Σ -coalgebra with $\delta(t, i) = t \upharpoonright i$ and $\ell(t) = t(\varepsilon)$. This is the final object in the category of Σ -coalgebras, because for every coalgebra (S, δ, ℓ) , there is a unique homomorphism $h : S \rightarrow C_\Sigma$ defined by $h(s)(\alpha) = \ell(\widehat{\delta}(s, \alpha))$.

2.3 Capsules and Representation

A *pointed Σ -coalgebra* is a Σ -coalgebra S with a distinguished state $s \in S$. A pointed Σ -coalgebra uniquely represents a Σ -coterm $h(s) \in C_\Sigma$, where $h : S \rightarrow C_\Sigma$ is the unique homomorphism. A Σ -coterm is *regular* if it has a finite representation.

A capsule $\langle e, \sigma \rangle$ is a syntactic representation of a pointed λ -coalgebra, and as such represents a unique closed λ -coterm. First α -convert so that all binding operators λx are distinct and disjoint from $\text{dom } \sigma$. The states of the coalgebra are the occurrences of subterms of terms in e and $\sigma(x)$

²Abstractions λx of arity 1, application \cdot of arity 2, variables and constants of arity 0.

If neither of these cases holds, then we say that that occurrence of x is *free* in $\langle e, \sigma \rangle$. Precapsules can have free variables, but capsules cannot.

It is important to note that scope does not extend through bindings in σ . For example, consider the capsule $\langle \lambda x.y, [y = \lambda z.x, x = 2] \rangle$. The free occurrence of x in $\lambda z.x$ is not bound to the λx in $\lambda x.y$, but rather to the value 2. The coalgebra represented by the capsule has three states and represents the closed cotermin $\lambda x.\lambda z.2$. For this reason, one cannot simply substitute $\sigma(y)$ for y in e without α -conversion. This is also reflected in the evaluation rules to be given in §4.1. In a capsule $\langle e, \sigma \rangle$, all free variables in e or $\sigma(y)$ are in $\text{dom } \sigma$, therefore bound; thus every capsule represents a closed cotermin.

For another example, consider the precapsule $\langle \lambda x.y, [y = \lambda z.x] \rangle$. This is not a capsule, because x is free in $\lambda z.x$ but is not in the domain of the valuation. Here the free occurrence of x in $\lambda z.x$ is also free in the precapsule. To define the corresponding coalgebra, we must first α -convert to avoid capture, giving $\langle \lambda u.y, [y = \lambda z.x] \rangle$. This corresponds to a three-state coalgebra representing the open cotermin $\lambda u.\lambda z.x$.

Capsules may be α -converted. Abstraction operators λx and the free variables bound to them may be renamed as usual, and it is not necessary to look beyond the term in which the abstraction occurs. Variables in $\text{dom } \sigma$ may also be renamed along with all free occurrences. Capsules that are α -equivalent represent the same value.

3 Scoping Issues

We motivate the results of §4 with an example illustrating how dynamic scoping arises from a naive implementation of lazy substitution and how capsules and closures remedy the situation.

3.1 The λ -Calculus

The oldest and simplest of all functional languages is the λ -calculus. In this system, a *state* is a closed λ -term, and *computation* consists of a sequence of β -reductions

$$(\lambda x.d) e \rightarrow d[x/e],$$

where $d[x/e]$ denotes the safe substitution of e for all free occurrences of x in d . *Safe substitution* means that bound variables in d may have to be renamed (α -converted) to avoid capturing free variables of the substituted term e .

For example, consider the closed λ -term $(\lambda y.(\lambda z.\lambda y.z) 4) \lambda x.y) 3) 2$. Evaluating this term in applicative order³, we get the following sequence of terms leading to the value 3:

$$(\lambda y.(\lambda z.\lambda y.z) 4) \lambda x.y) 3) 2 \rightarrow (\lambda z.\lambda y.z) 4) (\lambda x.3) 2 \rightarrow (\lambda y.(\lambda x.3) 4) 2 \rightarrow (\lambda x.3) 4 \rightarrow 3 \quad (1)$$

³Here *applicative order*, also known as *left-to-right call-by-value order*, refers to the order of evaluation in which the leftmost innermost redex is reduced first, except that redexes in the scope of binding operators λx are ineligible for reduction.

No α -conversion was necessary. In fact, it can be shown that no α -conversion is *ever* necessary with applicative-order evaluation of closed terms.

However, the λ -calculus is confluent, and we may choose a different order of evaluation; but an alternative order may require α -conversion. For example, the following reduction sequence is also valid:

$$(\lambda y.(\lambda z.\lambda y.z\ 4)\ \lambda x.y)\ 3\ 2 \rightarrow (\lambda y.\lambda w.(\lambda x.y)\ 4)\ 3\ 2 \rightarrow (\lambda w.(\lambda x.3)\ 4)\ 2 \rightarrow (\lambda x.3)\ 4 \rightarrow 3 \quad (2)$$

A change of bound variable was required in the first step to avoid capturing the free occurrence of y in $\lambda x.y$ substituted for z . Failure to do this results in the erroneous value 2:

$$(\lambda y.(\lambda z.\lambda y.z\ 4)\ \lambda x.y)\ 3\ 2 \rightarrow (\lambda y.\lambda y.(\lambda x.y)\ 4)\ 3\ 2 \rightarrow (\lambda y.(\lambda x.y)\ 4)\ 2 \rightarrow (\lambda x.2)\ 4 \rightarrow 2 \quad (3)$$

3.2 Dynamic Scoping

In the early development of functional programming, specifically with the language LISP, it was quickly determined that physical substitution is too inefficient because it requires copying. This led to the introduction of *environments*, used to effect lazy substitution. Instead of doing the actual substitution when performing a β -reduction, one can defer the substitution by saving it in an environment, then look up the value when needed.

An *environment* is a partial function $\sigma : \text{Var} \rightarrow \text{Irred}$ with finite domain. A *state* is a pair $\langle e, \sigma \rangle$, where e is the term to be evaluated and σ is an environment with bindings for the free variables in e . Environments need to be updated, which requires a *rebinding operator*

$$\sigma[x/e](y) = \begin{cases} e, & x = y, \\ \sigma(y), & x \neq y \end{cases}$$

Naively implemented, the rules are

$$\langle (\lambda x.d)\ e, \sigma \rangle \rightarrow \langle d, \sigma[x/e] \rangle \qquad \langle y, \sigma \rangle \rightarrow \langle \sigma(y), \sigma \rangle$$

where the first rule saves the deferred substitution in the environment and the second looks up the value. This is quite easy to implement. Moreover, it stands to reason that if β -reduction in applicative order does not require any α -conversions, then the lazy approach should not either. After all, the same terms are being substituted, just at a later time.

However, this is not the case. In the example above, we obtain the following sequence of states leading to the value 2:

$$\begin{array}{ll} (\lambda y.(\lambda z.\lambda y.z\ 4)\ \lambda x.y)\ 3\ 2 & [] \\ (\lambda z.\lambda y.z\ 4)\ (\lambda x.y)\ 2 & [y = 3] \\ (\lambda y.z\ 4)\ 2 & [y = 3, z = \lambda x.y] \\ z\ 4 & [y = 2, z = \lambda x.y] \\ (\lambda x.y)\ 4 & [y = 2, z = \lambda x.y] \\ y & [y = 2, z = \lambda x.y, x = 4] \\ 2 & [y = 2, z = \lambda x.y, x = 4] \end{array}$$

The issue is that the lazy approach fails to observe safe substitution. This example effectively performs the deferred substitutions in the order (3) without the change of bound variable. Nevertheless, this was the strategy adopted by early versions of LISP. It was not considered a bug but a feature and was called *dynamic scoping*.

3.3 Static Scoping with Closures

The semantics of evaluation was brought more in line with the λ -calculus with the introduction of *closures*, introduced in the language Scheme. Formally, a *closure* is defined as a pair $\{\lambda x.e, \sigma\}$, where the $\lambda x.e$ is a λ -abstraction and σ is a partial function from variables to values that is used to interpret the free variables of $\lambda x.e$. When a λ -abstraction is evaluated, it is paired with the environment σ at the point of the evaluation, and the value is the closure $\{\lambda x.e, \sigma\}$. Thus we have

$$\sigma : \text{Var} \rightarrow \text{Val} \qquad \text{Val} = \text{Const} + \text{Cl}$$

where Cl denotes the set of closures. We require that for a closure $\{\lambda x.e, \sigma\}$, $\text{FV}(\lambda x.e) \subseteq \text{dom } \sigma$. Note that the definitions of values and closures are mutually dependent.

The new reduction rules are

$$\langle \lambda x.d, \sigma \rangle \rightarrow \{\lambda x.d, \sigma\} \qquad \langle \{\lambda x.d, \sigma\} e, \tau \rangle \rightarrow \langle d, \sigma[x/e] \rangle \qquad \langle y, \sigma \rangle \rightarrow \sigma(y).$$

The second rule says that an application uses the context σ that was in effect when the closure was created, not the context τ of the call. Turning to our running example,

$$\begin{array}{ll} (\lambda y.(\lambda z.\lambda y.z \ 4) \ \lambda x.y) \ 3 \ 2 & [] \\ (\lambda z.\lambda y.z \ 4) \ (\lambda x.y) \ 2 & [y = 3] \\ (\lambda y.z \ 4) \ 2 & [y = 3, z = \{\lambda x.y, [y = 3]\}] \\ z \ 4 & [y = 2, z = \{\lambda x.y, [y = 3]\}] \\ \{\lambda x.y, [y = 3]\} \ 4 & [y = 2, z = \{\lambda x.y, [y = 3]\}] \\ (\lambda x.y) \ 4 & [y = 3] \\ y & [y = 3, x = 4] \\ 3 & [y = 3, x = 4] \end{array}$$

3.4 Static Scoping with Capsules

Closures correctly capture the semantics of β -reduction with safe substitution, but at the expense of introducing a rather involved combinatorial notion of state. Capsules allow us to revert to a more algebraic framework without losing the benefits of closures.

Capsules were defined formally in §2.1. The reduction rules for capsules are

$$\langle (\lambda x.e) \ v, \sigma \rangle \rightarrow \langle e[x/y], \sigma[y/v] \rangle \quad (y \text{ fresh}) \qquad \langle y, \sigma \rangle \rightarrow \langle \sigma(y), \sigma \rangle$$

The key difference is the introduction of the fresh variable y in the application rule. This is tantamount to performing an α -conversion on the parameter of a function just before applying it. Turning to our running example, we see that this approach gives the correct result.

$$\begin{array}{ll}
(\lambda y.(\lambda z.\lambda y.z\ 4)\ \lambda x.y)\ 3\ 2 & [] \\
(\lambda z.\lambda y.z\ 4)\ (\lambda x.y')\ 2 & [y' = 3] \\
(\lambda y.z'\ 4)\ 2 & [y' = 3, z' = \lambda x.y'] \\
z'\ 4 & [y' = 3, z' = \lambda x.y', y'' = 2] \\
(\lambda x.y')\ 4 & [y' = 3, z' = \lambda x.y', y'' = 2] \\
y' & [y' = 3, z' = \lambda x.y', y'' = 2, x' = 4] \\
3 & [y' = 3, z' = \lambda x.y', y'' = 2, x' = 4]
\end{array}$$

We prove soundness formally in §4.

4 Soundness

In this section we show that capsules correctly capture static scoping under applicative-order evaluation. We first show that capsules correctly model β -reduction in the λ -calculus with safe substitution.

4.1 Evaluation Rules for Capsules

Let d, e, \dots denote λ -terms and u, v, \dots irreducible λ -terms (λ -abstractions and constants). Variables are denoted x, y, \dots and constants c, f .

The small-step evaluation rules for capsules consist of reduction rules

$$\langle (\lambda x.e)\ v, \sigma \rangle \rightarrow \langle e[x/y], \sigma[y/v] \rangle \quad (y \text{ fresh}) \quad (4)$$

$$\langle f\ c, \sigma \rangle \rightarrow \langle f(c), \sigma \rangle \quad (5)$$

$$\langle y, \sigma \rangle \rightarrow \langle \sigma(y), \sigma \rangle \quad (6)$$

and context rules

$$\frac{\langle d, \sigma \rangle \xrightarrow{*} \langle d', \tau \rangle}{\langle d\ e, \sigma \rangle \xrightarrow{*} \langle d'\ e, \tau \rangle} \quad \frac{\langle e, \sigma \rangle \xrightarrow{*} \langle e', \tau \rangle}{\langle v\ e, \sigma \rangle \xrightarrow{*} \langle v\ e', \tau \rangle} \quad (7)$$

The reduction rules (4)–(6) identify three forms of redex: an application $(\lambda x.e)\ v$, an application $f\ c$ where f and c are constants, or a variable $y \in \text{dom}\ \sigma$. The context rules (7) uniquely identify a redex in a well-typed non-irreducible capsule according to an applicative-order reduction strategy.

The corresponding large-step rules are

$$\langle y, \sigma \rangle \rightarrow \langle \sigma(y), \sigma \rangle \quad (8)$$

$$\frac{\langle d, \sigma \rangle \xrightarrow{*} \langle f, \tau \rangle \quad \langle e, \tau \rangle \xrightarrow{*} \langle c, \rho \rangle}{\langle d e, \sigma \rangle \xrightarrow{*} \langle f(c), \rho \rangle} \quad (9)$$

$$\frac{\langle d, \sigma \rangle \xrightarrow{*} \langle \lambda x.a, \tau \rangle \quad \langle e, \tau \rangle \xrightarrow{*} \langle v, \rho \rangle \quad \langle a[x/y], \rho[y/v] \rangle \xrightarrow{*} \langle u, \pi \rangle}{\langle d e, \sigma \rangle \xrightarrow{*} \langle u, \pi \rangle} \quad (y \text{ fresh}) \quad (10)$$

These rules are best understood in terms of the interpreter they generate:

$$\begin{aligned} \text{Eval}(c, \sigma) &= \langle c, \sigma \rangle \\ \text{Eval}(\lambda x.e, \sigma) &= \langle \lambda x.e, \sigma \rangle \\ \text{Eval}(y, \sigma) &= \langle \sigma(y), \sigma \rangle \\ \text{Eval}(d e, \sigma) &= \text{let } \langle u, \tau \rangle = \text{Eval}(d, \sigma) \text{ in} \\ &\quad \text{let } \langle v, \rho \rangle = \text{Eval}(e, \tau) \text{ in} \\ &\quad \text{Apply}(u, v, \rho) \end{aligned} \quad (11)$$

$$\begin{aligned} \text{Apply}(f, c, \sigma) &= \langle f(c), \sigma \rangle \\ \text{Apply}(\lambda x.e, v, \sigma) &= \text{Eval}(e[x/y], \sigma[y/v]) \quad (y \text{ fresh}) \end{aligned} \quad (12)$$

4.2 β -Reduction

The small-step evaluation rules for β -reduction in applicative order are the same as for capsules, except we replace (4) with

$$\langle (\lambda x.e) v, \sigma \rangle \rightarrow \langle e[x/v], \sigma \rangle \quad (13)$$

(substitution instead of rebinding). The other rules (5)–(7) are the same. This makes sense even in the presence of cycles (recursive functions).

Note that the initial valuation σ persists unchanged throughout the computation. We might suppress it to simplify notation, giving

$$\begin{aligned} (\lambda x.e) v &\rightarrow e[x/v] & f c &\rightarrow f(c) & y &\rightarrow \sigma(y) \\ \frac{d \xrightarrow{*} d'}{(d e) \xrightarrow{*} (d' e)} & & \frac{e \xrightarrow{*} e'}{(v e) \xrightarrow{*} (v e')} & & \end{aligned}$$

However, it is still implicitly present, as it is needed to evaluate variables y .

The corresponding interpreter Eval_β is defined exactly like Eval except for rule (12), which we replace with

$$\text{Apply}_\beta(\lambda x.e, v, \sigma) = \text{Eval}_\beta(e[x/v], \sigma).$$

4.3 Soundness

Let S denote a sequential composition of rebinding operators $[y_1/v_1] \cdots [y_k/v_k]$, applied from left to right. Applied to a partial valuation $\sigma : \text{Var} \rightarrow \text{Irred}$, the operator S sequentially rebinds y_1 to v_1 , then y_2 to v_2 , and so on. The result is denoted σS . Formally, $\sigma(S[y/v]) = (\sigma S)[y/v]$.

To every rebinding operator $S = [y_1/v_1] \cdots [y_k/v_k]$ there corresponds a safe substitution operator $S^- = [y_k/v_k] \cdots [y_1/v_1]$, also applied from left to right. Applied to a λ -term e , S^- safely substitutes v_k for all free occurrences of y_k in e , then v_{k-1} for all free occurrences of y_{k-1} in $e[y_k/v_k]$, and so on. The result is denoted eS^- . Formally, $e(S^-[y/v]) = (eS^-)[y/v]$. Note that $(ST)^- = T^-S^-$.

If $S = [y_1/v_1] \cdots [y_k/v_k]$, we assume that y_i does not occur in v_j for $i \geq j$; however, y_i may occur in v_j if $i < j$. This means that if $\text{FV}(e) \subseteq \{y_1, \dots, y_k\}$ and $\text{FV}(v_j) \subseteq \{y_1, \dots, y_{j-1}\}$, $1 \leq j \leq k$, then eS^- is closed.

The following theorem establishes soundness of capsule evaluation with respect to β -reduction in the λ -calculus.

Theorem 4.1 $\text{Eval}_\beta(e, \sigma) = \langle v, \sigma \rangle$ if and only if there exist irreducible terms v_1, \dots, v_k, u and a rebinding operator $S = [y_1/v_1] \cdots [y_k/v_k]$, where y_1, \dots, y_k do not occur in e, v , or σ , such that $\text{Eval}(e, \sigma) = \langle u, \sigma S \rangle$ and $v = uS^-$.

Proof. We show the implication in both directions by induction on the number of steps in the evaluation. The result is trivially true for inputs of the form $\langle c, \sigma \rangle$, $\langle \lambda x.e, \sigma \rangle$, and $\langle \sigma(y), \sigma \rangle$, and this gives the basis of the induction.

For an input of the form $\langle d e, \sigma \rangle$, we show the implication in both directions. We first show that if $\text{Eval}(d e, \sigma)$ is defined, then so is $\text{Eval}_\beta(d e, \sigma)$, and the relationship between the two values is as described in the statement of the theorem. By definition of Eval , we have

$$\text{Eval}(d, \sigma) = (u, \sigma S) \qquad \text{Eval}(e, \sigma S) = (v, \sigma ST)$$

for some $S = [y_1/v_1] \cdots [y_m/v_m]$ and $T = [y_{m+1}/v_{m+1}] \cdots [y_n/v_n]$, where y_1, \dots, y_n are the fresh variables and v_1, \dots, v_n the irreducible terms bound to them in applications of the rule (12) during the evaluation of d and e . By the induction hypothesis, we have

$$\text{Eval}_\beta(d, \sigma) = \langle uS^-, \sigma \rangle \qquad \text{Eval}_\beta(e, \sigma S) = \langle vT^-, \sigma S \rangle.$$

Since the variables y_1, \dots, y_m do not occur in e , they are not accessed in its evaluation, thus

$$\text{Eval}_\beta(e, \sigma) = \langle vT^-, \sigma \rangle.$$

Also, since y_{m+1}, \dots, y_n do not occur in u and y_1, \dots, y_m do not occur in v , we have $uS^- = u(ST)^-$ and $vT^- = v(ST)^-$, thus

$$\text{Eval}_\beta(d, \sigma) = \langle u(ST)^-, \sigma \rangle \qquad \text{Eval}_\beta(e, \sigma) = \langle v(ST)^-, \sigma \rangle.$$

We thus have

$$\text{Eval}(d e, \sigma) = \text{Apply}(u, v, \sigma ST) \qquad \text{Eval}_\beta(d e, \sigma) = \text{Apply}_\beta(u(ST)^-, v(ST)^-, \sigma)$$

If u and v are constants, say $u = f$ and $v = c$, then

$$\text{Eval}(d e, \sigma) = \text{Apply}(f, c, \sigma ST) = \langle f(c), \sigma ST \rangle \quad \text{Eval}_\beta(d e, \sigma) = \text{Apply}_\beta(f, c, \sigma) = \langle f(c), \sigma \rangle,$$

and the implication holds. If u is a λ -abstraction, say $u = \lambda x.a$, then $u(ST)^- = \lambda x.a(ST)^-$. Then

$$\begin{aligned} a(ST)^-[x/v(ST)^-] &= a[x/v](ST)^- \\ &= a[x/y_{n+1}][y_{n+1}/v](ST)^- \\ &= a[x/y_{n+1}](ST[y_{n+1}/v])^-, \end{aligned}$$

therefore

$$\begin{aligned} \text{Eval}(d e, \sigma) &= \text{Apply}(\lambda x.a, v, \sigma ST) \\ &= \text{Eval}(a[x/y_{n+1}], \sigma ST[y_{n+1}/v]) \\ \text{Eval}_\beta(d e, \sigma) &= \text{Apply}_\beta(\lambda x.a(ST)^-, v(ST)^-, \sigma) \\ &= \text{Eval}_\beta(a(ST)^-[x/v(ST)^-], \sigma) \\ &= \text{Eval}_\beta(a[x/y_{n+1}](ST[y_{n+1}/v])^-, \sigma), \end{aligned}$$

and the implication holds in this case as well.

For the reverse implication, assume that $\text{Eval}_\beta(d e, \sigma)$ is defined. Let $\langle u, \sigma \rangle = \text{Eval}_\beta(d, \sigma)$ and $\langle v, \sigma \rangle = \text{Eval}_\beta(e, \sigma)$. By the induction hypothesis, there exist variables y_1, \dots, y_m and irreducible terms v_1, \dots, v_m and r such that

$$u = rS^- \quad \text{Eval}(d, \sigma) = \langle r, \sigma S \rangle,$$

where $S = [y_1/v_1] \cdots [y_m/v_m]$. We also have $\langle v, \sigma S \rangle = \text{Eval}_\beta(e, \sigma S)$, since the evaluation of e does not depend on the variables y_1, \dots, y_m . Again by the induction hypothesis, there exist variables y_{m+1}, \dots, y_n and irreducible terms v_{m+1}, \dots, v_n and s such that

$$v = sT^- = sT^-S^- = s(ST)^- \quad \text{Eval}(e, \sigma S) = \langle s, \sigma ST \rangle,$$

where $T = [y_{m+1}/v_{m+1}] \cdots [y_n/v_n]$. Then $ST = [y_1/v_1] \cdots [y_n/v_n]$ and

$$\text{Eval}_\beta(d e, \sigma) = \text{Apply}_\beta(u, v, \sigma) \quad \text{Eval}(d e, \sigma) = \text{Apply}(r, s, \sigma ST).$$

If u and v are constants, say $u = f$ and $v = c$, then $r = f$ and $s = c$. In this case we have

$$\text{Eval}_\beta(d e, \sigma) = \text{Apply}_\beta(f, c, \sigma) = \langle f(c), \sigma \rangle \quad \text{Eval}(d e, \sigma) = \text{Apply}(f, c, \sigma ST) = \langle f(c), \sigma ST \rangle,$$

and the implication holds. If u is a λ -abstraction, then $r = \lambda x.a$ and $u = \lambda x.aS^- = \lambda x.a(ST)^-$. In this case

$$\begin{aligned} a(ST)^-[x/s(ST)^-] &= a[x/s](ST)^- \\ &= a[x/y_{n+1}][y_{n+1}/s](ST)^- \\ &= a[x/y_{n+1}](ST[y_{n+1}/s])^-, \end{aligned}$$

thus

$$\begin{aligned}
\text{Eval}_\beta(d\ e, \sigma) &= \text{Apply}_\beta(\lambda x.a(ST)^-, v, \sigma) \\
&= \text{Eval}_\beta(a(ST)^-[x/s(ST)^-], \sigma) \\
&= \text{Eval}_\beta(a[x/y_{n+1}](ST[y_{n+1}/s])^-, \sigma), \\
\text{Eval}(d\ e, \sigma) &= \text{Apply}(\lambda x.a, s, \sigma ST) \\
&= \text{Eval}(a[x/y_{n+1}], \sigma ST[y_{n+1}/s]),
\end{aligned}$$

so the implication holds in this case as well. \square

4.4 Closure Conversion

In this section we demonstrate how to closure-convert a capsule and show that the transformation is sound with respect to the evaluation semantics of closures and capsules in applicative-order evaluation, provided variables are not mutable.

Closures do not work in the presence of mutable variables without introducing the further complication of references and indirection. This is because closures fix the environment once and for all when the closure is formed, whereas mutable variables allow the environment to be subsequently changed. An example is given by $(\lambda y.(\lambda x.y)\ (y := 4;y))\ 3$, for which capsules give 4 and closures 3; in the latter, the assignment has no effect.

Care must also be taken to implement updates nondestructively so as not to overwrite parameters and local variables of recursive procedures, an issue that is usually addressed at the implementation level. Again, the issue does not arise with capsules.

Even without indirection, the types of closures and closure environments are more involved than those of capsules. The definitions are mutually dependent and require a recursive type definition. The types are

$$\begin{aligned}
\text{Env} &= \text{Var} \rightarrow \text{Val} && \text{closure environments} \\
\text{Val} &= \text{Const} + \text{Cl} && \text{values} \\
\text{Cl} &= \lambda\text{-Abs} \times \text{Env} && \text{closures}
\end{aligned}$$

We use boldface for closure environments $\sigma : \text{Env}$ to distinguish them from the simpler capsule environments. Closures $\{\lambda x.e, \sigma\}$ must satisfy the additional requirement that $\text{FV}(\lambda x.e) \subseteq \text{dom } \sigma$.

A *state* is now a pair $\langle e, \sigma \rangle$, where $\text{FV}(e) \subseteq \text{dom } \sigma$, but the result of an evaluation is a Val. The

evaluation semantics for closures, expressed as an interpreter Eval_c , is

$$\begin{aligned}
\text{Eval}_c(c, \sigma) &= c \\
\text{Eval}_c(\lambda x.e, \sigma) &= \{\lambda x.e, \sigma\} \\
\text{Eval}_c(y, \sigma) &= \sigma(y) \\
\text{Eval}_c(d e, \sigma) &= \text{let } u = \text{Eval}_c(d, \sigma) \text{ in} \\
&\quad \text{let } v = \text{Eval}_c(e, \sigma) \text{ in} \\
&\quad \text{Apply}_c(u, v) \\
\text{Apply}_c(f, c) &= f(c) \\
\text{Apply}_c(\{\lambda x.a, \rho\}, v) &= \text{Eval}_c(a, \rho[x/v])
\end{aligned} \tag{14}$$

The types are

$$\text{Eval}_c : T_\lambda \times \text{Env} \rightarrow \text{Val} \qquad \text{Apply}_c : \text{Val} \times \text{Val} \rightarrow \text{Val}.$$

The correspondence with capsules becomes simpler to state if we modify the interpreter to α -convert the term $\lambda x.a$ to $\lambda y.a[x/y]$ just before applying it, where y is the fresh variable that would be chosen by the capsule interpreter. Accordingly, we replace (14) with

$$\text{Apply}_c(\{\lambda x.a, \rho\}, v) = \text{Eval}_c(a[x/y], \rho[y/v]) \quad (y \text{ fresh})$$

The corresponding large-step rules are

$$\langle c, \sigma \rangle \xrightarrow{c} c \qquad \langle \lambda x.e, \sigma \rangle \xrightarrow{c} \{\lambda x.e, \sigma\} \qquad \langle y, \sigma \rangle \xrightarrow{c} \sigma(y) \tag{15}$$

$$\frac{\langle d, \sigma \rangle \xrightarrow{c} f \quad \langle e, \sigma \rangle \xrightarrow{c} c}{\langle d e, \sigma \rangle \xrightarrow{c} f(c)} \tag{16}$$

$$\frac{\langle d, \sigma \rangle \xrightarrow{c} \{\lambda x.a, \rho\} \quad \langle e, \sigma \rangle \xrightarrow{c} v \quad \langle a[x/y], \rho[y/v] \rangle \xrightarrow{c} u}{\langle d e, \sigma \rangle \xrightarrow{c} u} \quad (y \text{ fresh}) \tag{17}$$

The closure-converted form of a capsule $\langle e, \sigma \rangle$ is $\langle e, \bar{\sigma} \rangle$, where

$$\bar{\sigma}(y) = \begin{cases} \{\sigma(y), \bar{\sigma}\}, & \text{if } \sigma(y) : \lambda\text{-Abs}, \\ \sigma(y), & \text{if } \sigma(y) : \text{Const}. \end{cases}$$

This definition is not circular, it is coinductive! In an OCaml-like language, the definition might look like

```

let rec  $\bar{\sigma} = \lambda y.$  match  $\sigma(y)$  with
| Const( $c$ )  $\rightarrow c$ 
|  $\lambda\text{-Abs}(\lambda x.e) \rightarrow \{\lambda x.e, \bar{\sigma}\}$ 

```

To state the relationship between capsules and closures, we define a binary relation \sqsubseteq on capsule environments, closure environments, and values. For capsule environments, define $\sigma \sqsubseteq \tau$ if $\text{dom } \sigma \subseteq \text{dom } \tau$ and for all $y \in \text{dom } \sigma$, $\sigma(y) = \tau(y)$. The definition for values and closure environments is by mutual coinduction: \sqsubseteq is defined to be the largest relation such that

- on closure environments, $\sigma \sqsubseteq \tau$ if
 - $\text{dom } \sigma \subseteq \text{dom } \tau$, and
 - for all $y \in \text{dom } \sigma$, $\sigma(y) \sqsubseteq \tau(y)$; and
- on values, $u \sqsubseteq v$ if either
 - u and v are constants and $u = v$; or
 - $u = \{\lambda x.e, \rho\}$, $v = \{\lambda x.e, \pi\}$, and $\rho \sqsubseteq \pi$.

Lemma 4.2 *The relation \sqsubseteq is transitive.*

Proof. This is obvious for capsule environments.

For closure environments and values, we proceed by coinduction. Suppose $\sigma \sqsubseteq \tau \sqsubseteq \rho$. Then $\text{dom } \sigma \subseteq \text{dom } \tau \subseteq \text{dom } \rho$, so $\text{dom } \sigma \subseteq \text{dom } \rho$, and for all $y \in \text{dom } \sigma$, $\sigma(y) \sqsubseteq \tau(y) \sqsubseteq \rho(y)$, therefore $\sigma(y) \sqsubseteq \rho(y)$ by the transitivity of \sqsubseteq on values.

For values, suppose $u \sqsubseteq v \sqsubseteq w$. If $u = c$, then $v = c$ and $w = c$. If $u = \{\lambda x.e, \sigma\}$, then $v = \{\lambda x.e, \tau\}$ and $w = \{\lambda x.e, \rho\}$ and $\sigma \sqsubseteq \tau \sqsubseteq \rho$, therefore $\sigma \sqsubseteq \rho$ by the transitivity of \sqsubseteq on closure environments. \square

Lemma 4.3 *Closure conversion is monotone with respect to \sqsubseteq . That is, if $\sigma \sqsubseteq \tau$, then $\bar{\sigma} \sqsubseteq \bar{\tau}$.*

Proof. We have $\text{dom } \bar{\sigma} = \text{dom } \sigma \subseteq \text{dom } \tau = \text{dom } \bar{\tau}$. Moreover, for $y \in \text{dom } \sigma$,

$$\begin{aligned} \bar{\sigma}(y) &= \begin{cases} \{\lambda x.e, \bar{\sigma}\}, & \text{if } \sigma(y) = \lambda x.e, \\ c, & \text{if } \sigma(y) = c \end{cases} \\ &= \begin{cases} \{\lambda x.e, \bar{\sigma}\}, & \text{if } \tau(y) = \lambda x.e, \\ c, & \text{if } \tau(y) = c \end{cases} \\ &\sqsubseteq \begin{cases} \{\lambda x.e, \bar{\tau}\}, & \text{if } \tau(y) = \lambda x.e, \\ c, & \text{if } \tau(y) = c \end{cases} \\ &= \bar{\tau}(y). \end{aligned}$$

The \sqsubseteq step in the above reasoning is by the coinduction hypothesis. \square

Define a map $V : \text{Cap} \rightarrow \text{Val}$ on irreducible capsules as follows:

$$V(\lambda x.a, \sigma) = \{\lambda x.a, \bar{\sigma}\} \qquad V(c, \sigma) = c. \qquad (18)$$

Lemma 4.4 $\bar{\sigma}(y) = V(\sigma(y), \sigma)$.

Proof.

$$\begin{aligned} \bar{\sigma}(y) &= \begin{cases} \{\lambda x.e, \bar{\sigma}\}, & \text{if } \sigma(y) = \lambda x.e, \\ c & \text{if } \sigma(y) = c \end{cases} \\ &= \begin{cases} V(\lambda x.e, \sigma), & \text{if } \sigma(y) = \lambda x.e, \\ V(c, \sigma) & \text{if } \sigma(y) = c \end{cases} \\ &= V(\sigma(y), \sigma). \end{aligned}$$

□

Lemma 4.5 *If $y \notin \text{dom } \sigma$, then $\overline{\sigma[y/V(v, \sigma)]} \sqsubseteq \overline{\sigma[y/v]}$.*

Proof. By Lemma 4.4,

$$\overline{\sigma[y/v]}(y) = V(\sigma[y/v](y), \sigma[y/v]) = V(v, \sigma[y/v]). \quad (19)$$

If $y \notin \text{dom } \sigma$, then

$$\overline{\sigma[y/V(v, \sigma)]} \sqsubseteq \overline{\sigma[y/v][y/V(v, \sigma)]} \sqsubseteq \overline{\sigma[y/v][y/V(v, \sigma[y/v])]} = \overline{\sigma[y/v]},$$

the first two inequalities by Lemma 4.3 and the last equation by (19). □

Lemma 4.6 *If $\sigma \sqsubseteq \tau$, then $\text{Eval}_c(e, \sigma)$ exists if and only if $\text{Eval}_c(e, \tau)$ does, and $\text{Eval}_c(e, \sigma) \sqsubseteq \text{Eval}_c(e, \tau)$. Moreover, they are derivable by the same large-step proofs.*

Proof. We proceed by induction on the proof tree under the large-step rules (15)–(17). For the single-step rules (15), we have

$$\begin{aligned} \text{Eval}_c(c, \sigma) &= c = \text{Eval}_c(c, \tau) \\ \text{Eval}_c(\lambda x.a, \sigma) &= \{\lambda x.a, \sigma\} \sqsubseteq \{\lambda x.a, \tau\} = \text{Eval}_c(\lambda x.a, \tau) \\ \text{Eval}_c(y, \sigma) &= \sigma(y) \sqsubseteq \tau(y) = \text{Eval}_c(y, \tau). \end{aligned}$$

For the rule (16), $\langle d e, \sigma \rangle \xrightarrow{*}_c f(c)$ is derivable by an application of (16) iff $\langle d, \sigma \rangle \xrightarrow{*}_c f$ and $\langle e, \sigma \rangle \xrightarrow{*}_c c$ are derivable by smaller proofs. Similarly, $\langle d e, \tau \rangle \xrightarrow{*}_c f(c)$ is derivable by an application of (16) iff $\langle d, \tau \rangle \xrightarrow{*}_c f$ and $\langle e, \tau \rangle \xrightarrow{*}_c c$ are derivable by smaller proofs. By the induction hypothesis, $\langle d, \sigma \rangle \xrightarrow{*}_c f$ and $\langle d, \tau \rangle \xrightarrow{*}_c f$ are derivable by the same proof, and similarly $\langle e, \sigma \rangle \xrightarrow{*}_c c$ and $\langle e, \tau \rangle \xrightarrow{*}_c c$ are derivable by the same proof.

Finally, for the rule (17), $\langle d e, \sigma \rangle \xrightarrow{*}_c u_1$ is derivable by an application of (17) iff $\langle d, \sigma \rangle \xrightarrow{*}_c \{\lambda x.a, \rho_1\}$, $\langle e, \sigma \rangle \xrightarrow{*}_c v_1$, and $\langle a[x/y], \rho_1[y/v_1] \rangle \xrightarrow{*}_c u_1$ are derivable by smaller proofs. Similarly, $\langle d e, \tau \rangle \xrightarrow{*}_c u_2$ is derivable by an application of (17) iff $\langle d, \tau \rangle \xrightarrow{*}_c \{\lambda x.a, \rho_2\}$, $\langle e, \tau \rangle \xrightarrow{*}_c v_2$, and $\langle a[x/y], \rho_2[y/v_2] \rangle \xrightarrow{*}_c u_2$ are derivable by smaller proofs. By the induction hypothesis, $\langle d, \sigma \rangle \xrightarrow{*}_c \{\lambda x.a, \rho_1\}$ and $\langle d, \tau \rangle \xrightarrow{*}_c \{\lambda x.a, \rho_2\}$ are derivable by the same proof, and $\rho_1 \sqsubseteq \rho_2$. Similarly, $\langle e, \sigma \rangle \xrightarrow{*}_c v_1$ and $\langle e, \tau \rangle \xrightarrow{*}_c v_2$ are derivable by the same proof, and $v_1 \sqsubseteq v_2$. It follows that $\rho_1[y/v_1] \sqsubseteq \rho_2[y/v_2]$. Again by the induction hypothesis, $\langle a[x/y], \rho_1[y/v_1] \rangle \xrightarrow{*}_c u_1$ and $\langle a[x/y], \rho_2[y/v_2] \rangle \xrightarrow{*}_c u_2$ are derivable by the same proof, and $u_1 \sqsubseteq u_2$. □

The following theorem establishes the soundness of closure conversion for capsules.

Theorem 4.7 *$\text{Eval}(e, \sigma)$ exists if and only if $\text{Eval}_c(e, \overline{\sigma})$ does, and $\text{Eval}_c(e, \overline{\sigma}) \sqsubseteq V(\text{Eval}(e, \sigma))$. Moreover, they are derivable by isomorphic large-step proofs under the obvious correspondence between the large-step rules of both systems.⁴*

Proof. We proceed by induction on the proof tree under the large-step rules. The proof is similar to the proof of Lemma 4.6. We write $\xrightarrow{*}_c$ for the derivability relation under the large-step rules (15)–(17) for closures to distinguish them from the corresponding large-step rules (8)–(10) for capsules, which we continue to denote by $\xrightarrow{*}$.

⁴For this purpose, the definition of V in (18) can be viewed as a pair of proof rules corresponding to the first two rules of (15).

For the single-step rules (15), we have

$$\begin{aligned}\text{Eval}_c(c, \bar{\sigma}) &= c = V(\text{Eval}(c, \sigma)) \\ \text{Eval}_c(\lambda x.a, \bar{\sigma}) &= \{\lambda x.a, \bar{\sigma}\} = V(\lambda x.a, \sigma) = V(\text{Eval}(\lambda x.a, \sigma)) \\ \text{Eval}_c(y, \bar{\sigma}) &= \bar{\sigma}(y) = V(\sigma(y), \sigma) = V(\text{Eval}(y, \sigma)).\end{aligned}$$

The last line uses Lemma 4.4.

Consider the corresponding rules (9) and (16). A conclusion $\langle d e, \bar{\sigma} \rangle \xrightarrow{*}_c f(c)$ is derivable by an application of (16) iff $\langle d, \bar{\sigma} \rangle \xrightarrow{*}_c f$ and $\langle e, \bar{\sigma} \rangle \xrightarrow{*}_c c$ are derivable by smaller proofs. Similarly, $\langle d e, \sigma \rangle \xrightarrow{*} \langle f(c), \rho \rangle$ is derivable by an application of (9) iff $\langle d, \sigma \rangle \xrightarrow{*} \langle f, \sigma S \rangle$ and $\langle e, \sigma S \rangle \xrightarrow{*} \langle c, \sigma ST \rangle$ are derivable by smaller proofs.

By the induction hypothesis, $\langle d, \bar{\sigma} \rangle \xrightarrow{*}_c f = V(f, \sigma S)$ and $\langle d, \sigma \rangle \xrightarrow{*} \langle f, \sigma S \rangle$ are derivable by isomorphic proofs. By Lemma 4.6, $\langle e, \bar{\sigma} \rangle \xrightarrow{*}_c c$ and $\langle e, \bar{\sigma S} \rangle \xrightarrow{*}_c c$ are derivable by the same proof. Again by the induction hypothesis, $\langle e, \bar{\sigma S} \rangle \xrightarrow{*}_c c$ and $\langle e, \sigma S \rangle \xrightarrow{*} \langle c, \sigma ST \rangle$ are derivable by isomorphic proofs, therefore so are $\langle e, \bar{\sigma} \rangle \xrightarrow{*}_c c = V(c, \sigma ST)$ and $\langle e, \sigma S \rangle \xrightarrow{*} \langle c, \sigma ST \rangle$.

Finally, consider the corresponding rules (10) and (17). A conclusion $\langle d e, \bar{\sigma} \rangle \xrightarrow{*}_c u$ is derivable by an application of (17) iff for some $\lambda x.a, \rho$, and v ,

$$\langle d, \bar{\sigma} \rangle \xrightarrow{*}_c \{\lambda x.a, \rho\} \quad \langle e, \bar{\sigma} \rangle \xrightarrow{*}_c v \quad \langle a[x/y], \rho[y/v] \rangle \xrightarrow{*}_c u$$

are derivable by smaller proofs. Similarly, $\langle d e, \sigma \rangle \xrightarrow{*} \langle t, \tau \rangle$ is derivable by an application of (10) iff for some $\lambda z.b, S, T$, and w ,

$$\langle d, \sigma \rangle \xrightarrow{*} \langle \lambda z.b, \sigma S \rangle \quad \langle e, \sigma S \rangle \xrightarrow{*} \langle w, \sigma ST \rangle \quad \langle b[z/y], \sigma ST[y/w] \rangle \xrightarrow{*} \langle t, \tau \rangle$$

are derivable by smaller proofs.

By the induction hypothesis, $\langle d, \bar{\sigma} \rangle \xrightarrow{*}_c \{\lambda x.a, \rho\}$ and $\langle d, \sigma \rangle \xrightarrow{*} \langle \lambda z.b, \sigma S \rangle$ are derivable by isomorphic proofs, and $\{\lambda x.a, \rho\} \sqsubseteq V(\lambda z.b, \sigma S) = \{\lambda z.b, \bar{\sigma S}\}$, therefore $\lambda x.a = \lambda z.b$ and $\rho \sqsubseteq \bar{\sigma S} \sqsubseteq \bar{\sigma ST}$.

By Lemmas 4.3 and 4.6, for some v' , $\langle e, \bar{\sigma} \rangle \xrightarrow{*}_c v$ and $\langle e, \bar{\sigma S} \rangle \xrightarrow{*}_c v'$ are derivable by the same proof, and $v \sqsubseteq v'$. Again by the induction hypothesis, $\langle e, \bar{\sigma S} \rangle \xrightarrow{*}_c v'$ and $\langle e, \sigma S \rangle \xrightarrow{*} \langle w, \sigma ST \rangle$ are derivable by isomorphic proofs, and $v' \sqsubseteq V(w, \sigma ST)$. By transitivity, $\langle e, \bar{\sigma} \rangle \xrightarrow{*}_c v$ and $\langle e, \sigma S \rangle \xrightarrow{*} \langle w, \sigma ST \rangle$ are derivable by isomorphic proofs, and $v \sqsubseteq V(w, \sigma ST)$. By Lemma 4.5,

$$\rho[y/v] \sqsubseteq \bar{\sigma ST}[y/V(w, \sigma ST)] \sqsubseteq \bar{\sigma ST}[y/w].$$

Again by Lemma 4.6, for some u' , $\langle a[x/y], \rho[y/v] \rangle \xrightarrow{*}_c u$ and $\langle a[x/y], \bar{\sigma ST}[y/w] \rangle \xrightarrow{*}_c u'$ are derivable by the same proof, and $u \sqsubseteq u'$; and again by the induction hypothesis, $\langle a[x/y], \bar{\sigma ST}[y/w] \rangle \xrightarrow{*}_c u'$ and $\langle a[x/y], \sigma ST[y/w] \rangle \xrightarrow{*} \langle t, \tau \rangle$ are derivable by isomorphic proofs, and $u' \sqsubseteq V(t, \tau)$. By transitivity, $\langle a[x/y], \rho[y/v] \rangle \xrightarrow{*}_c u$ and $\langle a[x/y], \sigma ST[y/w] \rangle \xrightarrow{*} \langle t, \tau \rangle$ are derivable by isomorphic proofs, and $u \sqsubseteq V(t, \tau)$. \square

5 A Functional/Imperative Language

In this section we give an operational semantics for a simply-typed higher-order functional and imperative language with mutable bindings.

5.1 Expressions

Expressions $\text{Exp} = \{d, e, \dots\}$ contain both functional and imperative features. There is an unlimited supply of *variables* x, y, \dots of all (simple) types, as well as constants f, c, \dots for primitive values. In addition, there are functional features

- λ -abstraction $\lambda x.e$
- application $(d e),$

imperative features

- assignment $x := e$
- composition $d;e$
- conditional $\text{if } b \text{ then } d \text{ else } e$
- repeat loop $\text{repeat } e \text{ until } b,$

and syntactic sugar

- let $x = d$ in e $(\lambda x.e) d$
- let rec $f = g$ in e $\text{let } f = h \text{ in } f := g; e$

where h is any term of the appropriate type.

5.2 Types

Types are just simple types built inductively from the base types and a type constructor \rightarrow representing *partial* functions. The typing rules are:

$$\frac{x : \alpha \quad e : \beta}{\lambda x.e : \alpha \rightarrow \beta} \qquad \frac{d : \alpha \rightarrow \beta \quad e : \alpha}{(d e) : \beta} \qquad \frac{d : \alpha \quad e : \beta}{d;e : \beta}$$

$$\frac{b : \text{bool} \quad d : \alpha \quad e : \alpha}{\text{if } b \text{ then } d \text{ else } e : \alpha} \qquad \frac{b : \text{bool} \quad e : \alpha}{\text{repeat } e \text{ until } b : \alpha} \qquad \frac{x : \alpha \quad e : \alpha}{x := e : \alpha}$$

5.3 Evaluation

A *value* is the equivalence class of an irreducible capsule modulo bisimilarity and α -conversion; equivalently, the λ -coterms represented by the capsule modulo α -conversion.

A program determines a binary relation on capsules. The functional features are interpreted by the rules of §4.1. Assignment is interpreted by the following large-step and small-step rules, respectively:

$$\frac{\langle e, \sigma \rangle \xrightarrow{*} \langle v, \tau \rangle}{\langle x := e, \sigma \rangle \xrightarrow{*} \langle v, \tau[x/v] \rangle} \quad (x \in \text{dom } \sigma) \qquad \langle x := v, \tau \rangle \rightarrow \langle v, \tau[x/v] \rangle \quad (x \in \text{dom } \tau)$$

The remaining imperative constructs are defined by the following large-step rules.

$$\frac{\langle d, \sigma \rangle \xrightarrow{*} \langle u, \rho \rangle \quad \langle e, \rho \rangle \xrightarrow{*} \langle v, \tau \rangle}{\langle d; e, \sigma \rangle \xrightarrow{*} \langle v, \tau \rangle}$$

$$\frac{\langle b, \sigma \rangle \xrightarrow{*} \langle \text{true}, \rho \rangle \quad \langle d, \rho \rangle \xrightarrow{*} \langle v, \tau \rangle}{\langle \text{if } b \text{ then } d \text{ else } e, \sigma \rangle \xrightarrow{*} \langle v, \tau \rangle} \quad \frac{\langle b, \sigma \rangle \xrightarrow{*} \langle \text{false}, \rho \rangle \quad \langle e, \rho \rangle \xrightarrow{*} \langle v, \tau \rangle}{\langle \text{if } b \text{ then } d \text{ else } e, \sigma \rangle \xrightarrow{*} \langle v, \tau \rangle}$$

$$\frac{\langle e, \sigma \rangle \xrightarrow{*} \langle v, \rho \rangle \quad \langle b, \rho \rangle \xrightarrow{*} \langle \text{true}, \tau \rangle}{\langle \text{repeat } e \text{ until } b, \sigma \rangle \xrightarrow{*} \langle v, \tau \rangle} \quad \frac{\langle e; b, \sigma \rangle \xrightarrow{*} \langle \text{false}, \rho \rangle \quad \langle \text{repeat } e \text{ until } b, \rho \rangle \xrightarrow{*} \langle v, \tau \rangle}{\langle \text{repeat } e \text{ until } b, \sigma \rangle \xrightarrow{*} \langle v, \tau \rangle}$$

5.4 Garbage Collection

A *monomorphism* $h : \langle d, \sigma \rangle \rightarrow \langle e, \tau \rangle$ is an injective map $h : \text{dom } \sigma \rightarrow \text{dom } \tau$ such that

- $\tau(h(x)) = h(\sigma(x))$ for all $x \in \text{dom } \sigma$, where $h(e) = e[x/h(x)]$ (safe substitution); and
- $h(d) = e$.

The collection of monomorphic preimages of a given capsule contains an initial object that is unique up to α -conversion. This is the *garbage collected* version of the capsule.

6 Conclusion

We have presented *capsules* as a unified algebraic representation of state for higher-order functional and imperative programs. We believe that capsules are a fundamental construct that can aid in understanding and formalizing the semantics of languages with both functional and imperative features, including mutable bindings.

Capsules are mathematically simpler than closures and correctly model lexical scope without auxiliary data constructs, even in the presence of recursion and mutable variables. Capsules form a natural coalgebraic extension of the λ -calculus, and we have shown how coalgebraic techniques can be brought to bear on arguments involving state. We have shown that capsule evaluation is faithful to β -reduction with safe substitution in the λ -calculus. We have shown how to closure-convert capsules, and we have proved soundness of the transformation in the absence of assignments. Finally, we have shown how capsules can be used to give a natural operational semantics to a higher-order functional and imperative language with mutable bindings.

References

- [1] I. Mason and C. Talcott, “Equivalence in functional languages with effects,” 1991.

- [2] —, “Programming, transforming, and proving with function abstractions and memories.”
- [3] —, “Axiomatizing operational equivalence in the presence of side effects,” in *Fourth Annual Symposium on Logic in Computer Science. IEEE*. IEEE Computer Society Press, 1989, pp. 284–293.
- [4] M. Felleisen and R. Hieb, “The revised report on the syntactic theories of sequential control and state,” *Theoretical Computer Science*, vol. 103, pp. 235–271, 1992.
- [5] K. Aboul-Hosn, “Programming with private state,” Honors Thesis, The Pennsylvania State University, December 2001. [Online]. Available: <http://www.cs.cornell.edu/%7Ekamal/thesis.pdf>
- [6] E. Moggi, “Notions of computation and monads,” *Information and Computation*, vol. 93, no. 1, 1991.
- [7] R. Milner and C. Strachey, *A Theory of Programming Language Semantics*. New York, NY, USA: Halsted Press, 1977.
- [8] D. Scott, “Mathematical concepts in programming language semantics,” in *Proc. 1972 Spring Joint Computer Conferences*. Montvale, NJ: AFIPS Press, 1972, pp. 225–34.
- [9] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge, MA, USA: MIT Press, 1981.
- [10] J. Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot, “The semantics of local storage, or what makes the free-list free?” in *Proc. 11th ACM Symp. Principles of Programming Languages (POPL’84)*, New York, NY, USA, 1984, pp. 245–257.
- [11] K. Aboul-Hosn and D. Kozen, “Relational semantics for higher-order programs,” in *Proc. 8th Int. Conf. Mathematics of Program Construction (MPC’06)*, ser. Lecture Notes in Computer Science, T. Uustalu, Ed., vol. 4014. Springer, July 2006, pp. 29–48.
- [12] —, “Local variable scoping and Kleene algebra with tests,” *J. Log. Algebr. Program.*, 2007, doi: 10.1016/j.jlap.2007.10.007.
- [13] H. P. Barendregt and J. W. Klop, *Applications of Infinitary Lambda Terms*, to appear.
- [14] J. W. Klop and R. C. de Vrijer, “Infinitary normalization,” in *We Will Show Them: Essays in Honour of Dov Gabbay*, S. Artemov, H. Barringer, A. S. d’Avila Garcez, L. C. Lamb, and J. Woods, Eds. College Publications, 2005, vol. 2, pp. 169–192.
- [15] “Scope (programming),” Wikipedia. [Online]. Available: [http://en.wikipedia.org/wiki/Scope_\(programming\)](http://en.wikipedia.org/wiki/Scope_(programming))
- [16] M. Abadi and L. Cardelli, *A Theory of Objects*. Springer, 1996.
- [17] I. A. Mason and C. L. Talcott, “References, local variables and operational reasoning,” in *Seventh Annual Symposium on Logic in Computer Science*. IEEE, 1992, pp. 186–197. [Online]. Available: <http://www-formal.stanford.edu/MT/92lics.ps.Z>
- [18] A. M. Pitts, “Operational semantics and program equivalence,” INRIA Sophia Antipolis, Tech. Rep., 2000, lectures at the International Summer School On Applied

Semantics, APPSEM 2000, Caminha, Minho, Portugal, September 2000. [Online]. Available: <http://www.springerlink.com/media/1f99vvygyh3ygrykklby/contributions/1/%w/f/6/lwf6r3jxn7a2lkq0.pdf>

- [19] A. M. Pitts and I. D. B. Stark, "Observable properties of higher order functions that dynamically create local names, or what's new?" in *MFCs*, ser. Lecture Notes in Computer Science, A. M. Borzyszkowski and S. Sokolowski, Eds., vol. 711. Springer, 1993, pp. 122–141.
- [20] A. M. Pitts, "Operationally-based theories of program equivalence," in *Semantics and Logics of Computation*, ser. Publications of the Newton Institute, P. Dybjer and A. M. Pitts, Eds. Cambridge University Press, 1997, pp. 241–298. [Online]. Available: <http://www.cs.tau.ac.il/~nachumd/formal/exam/pitts.pdf>
- [21] A. M. Pitts and I. D. B. Stark, "Operational reasoning in functions with local state," in *Higher Order Operational Techniques in Semantics*, A. D. Gordon and A. M. Pitts, Eds. Cambridge University Press, 1998, pp. 227–273. [Online]. Available: <http://homepages.inf.ed.ac.uk/stark/operfl.pdf>
- [22] S. Abramsky, K. Honda, and G. McCusker, "A fully abstract game semantics for general references," in *LICS '98: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 334–344.
- [23] J. Laird, "A game semantics of local names and good variables." in *FoSSaCS*, ser. Lecture Notes in Computer Science, I. Walukiewicz, Ed., vol. 2987. Springer, 2004, pp. 289–303.
- [24] S. Abramsky and G. McCusker, "Linearity, sharing and state: a fully abstract game semantics for idealized ALGOL with active expressions." *Electr. Notes Theor. Comput. Sci.*, vol. 3, 1996.