Journal of Automata, Languages and Combinatorics **17** (2012) 2–4, 185–204

COMPUTING WITH CAPSULES

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

and

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

ABSTRACT

Capsules provide an algebraic representation of the state of a computation in higher-order functional and imperative languages. A capsule is essentially a finite coalgebraic representation of a regular closed λ-coterm. One can give an operational semantics based on capsules for a higher-order programming language with functional and imperative features, including mutable bindings. Static (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 fixpoint combinators.

Keywords: capsules, semantics, functional programming, imperative programming

1. Introduction

Capsules provide an algebraic representation of the state of a computation in higher-order functional and imperative programming languages. They conservatively extend the classical λ-calculus with mutable variables and assignment, enabling the construction of certain regular coterms (infinite terms) representing recursive functions without the need for fixpoint combinators. They have a well-defined statically-scoped evaluation semantics, are typable with simple types, and are Turing complete.

Representations of state have been studied in the past by many authors. Approaches include syntactic theories of control and state [11, 12], the semantics of local storage [14], functional languages with effects [22, 23, 24], monads [28], closure structures [3, 4, 5] and denotational semantics [27, 35, 36]. Capsules provide a purely algebraic alternative in that no combinatorial structures are needed. Perhaps the most important aspect of capsules is that static scoping and local variables are captured without the need for closures. Cumbersome combinatorial machinery such as

heaps, stores, stacks, and pointers are replaced with the single mathematical concept of variable binding. Nevertheless, capsules are equally expressive and represent the same data dependencies and liveness structure. In a sense, capsules are to closures what graphs are to their adjacency list representations.

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-coterm (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 [8, 18]. 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. 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 both functional and imperative features, including recursion and mutable variables, and this is one of the primary motivations of this work. All operations of interest are typable with simple types. Recursive functions are constructed directly using *Landin's knot* [21] without the need for fixpoint combinators, which involve self-application and are 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 construction of recursive functions with Landin's knot is direct and simply typable, and corresponds more closely to 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 and closer to implementations.

Dynamic scoping, which was the scoping discipline in early versions of LISP and Python, and which still exists in many languages today, can be regarded as an 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, the λ-calculus with β-reduction and safe substitution is statically scoped. Both capsules and closures provide static scoping, but capsules do so without any extra combinatorial machinery. Moreover, capsules work correctly in the presence of mutable variables, whereas closures, naively implemented, do not (a counterexample is given in §4.4). To correctly handle mutable variables, closures require some form of indirection, and care must be taken to perform updates nondestructively. The connection between closures and capsules in the presence of mutable variables has been investigated by the first author [15]. Capsules have also been applied to the study of separation logic [17].

Capsules provide a common framework for representing the global state of computation for both functional and imperative programs. Valuations of mutable variables used in the semantics of imperative programs and closure structures used in the operational semantics of functional programs can be simulated. Capsules also allow a clean mathematical 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.

There is much previous work on reasoning about references and local state [12, 25, 30, 31, 32, 33]. State is typically modeled by some form of heap from which storage locations can be allocated and deallocated [14, 22, 23, 24, 27, 35, 36]. Others use game semantics to reason about local state [6, 7, 20]. Moggi [28] uses monads to model state. Our approach is most closely related to the work of Mason and Talcott [22, 23, 24], Felleisen and Hieb [12], and especially to the syntactic theories of control and state of Felleisen, Findler, and Flatt [11]. Abadi, Cardelli, Curien and Lévy study substitutions explicitly [2], while Curien develops a calculus based on closures [10]. Moran and Sands develop an abstract machine to handle the call-by-need λ-calculus [29]. Objects can be modeled as collections of mutable bindings, as for example in the ς-calculus of Abadi and Cardelli [1]. Here we have avoided the introduction of mutable datatypes other than λ-terms in order to develop the theory in its simplest form and to emphasize that no auxiliary datatypes are needed to provide a basic operational semantics for a statically-scoped higher-order language with functional and imperative features.

This paper is organized as follows. In §2, we give formal definitions of capsules. In §3, we give a detailed motivating example comparing how closures and capsules deal with scoping issues. In §4 we prove two theorems. The first (Theorem 1) establishes that capsule evaluation faithfully models β-reduction in the λ-calculus with safe substitution. The second (Theorem 7) defines closure conversion for capsules and proves soundness of the translation, provided there is no variable assignment. Taken together, these two theorems establish that closures also correctly model β-reduction in the λ-calculus with safe substitution. The same results hold in the presence of assignment, but the definition of closures must be extended; the definition of capsules remains the same [15]. 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., $3 : \mathsf{int}$, $\mathsf{true} : \mathsf{bool}$, $+ : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$, $\leq : \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$). The set of λ-abstractions is denoted $\lambda\text{-}\mathsf{Abs}$ 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 $\mathsf{Irred} = \lambda\text{-}\mathsf{Abs} + \mathsf{Const}$. Note that variables x are not irreducible.

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

(i) $\mathsf{FV}(e) \subseteq \mathsf{dom}\,\sigma$

(ii) if $x \in \mathsf{dom}\,\sigma$, then $\mathsf{FV}(\sigma(x)) \subseteq \mathsf{dom}\,\sigma$.

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 $\sigma(f) = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \cdot f(n-1)$.

2.2. Scope, Free and Bound Variables

Let $\langle e, \sigma \rangle$ be a capsule and let d be either e or $\sigma(y)$ for some $y \in \mathsf{dom}\,\sigma$. The *scope* of an occurrence of a binding operator λx in d is its scope in the λ-term d as normally defined.

Consider an occurrence of a variable x in d. The closure conditions (i) and (ii) of §2.1 ensure that one of the following two conditions holds:

- that occurrence of x falls in the scope of a binding operator λx in d, in which case it is bound to the innermost binding operator λx in d in whose scope it lies; or
- it is free in d, but $x \in \mathsf{dom}\,\sigma$, in which case it is bound by σ to the value $\sigma(x)$.

Thus every variable x in a capsule is essentially bound. These conditions thus preclude catastrophic failure due to access of unbound variables.

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 term $\lambda x.\lambda z.2$. 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 $\mathsf{dom}\,\sigma$, therefore bound to a value; thus every capsule represents a closed coterm.

The term *α-conversion* refers to the renaming of bound variables. With a capsule $\langle e, \sigma \rangle$, this can happen in two ways. The traditional form maps a subterm $\lambda x.d$ to $\lambda y.d[x/y]$, provided y would not be captured in d. One can also rename a variable $x \in \mathsf{dom}\,\sigma$ and all free occurrences of x in e and $\sigma(z)$ for $z \in \mathsf{dom}\,\sigma$ to y, provided $y \notin \mathsf{dom}\,\sigma$ already and y would not be captured.

3. Scoping Issues

We motivate the results of Section 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 (shallow) applicative order[1], we get the following sequence of terms leading to the value 3:

$$\begin{aligned}(\lambda y.(\lambda z.\lambda y.z\ 4)\ \lambda x.y)\ 3\ 2 &\to (\lambda z.\lambda y.z\ 4)\ (\lambda x.3)\ 2\\ &\to (\lambda y.(\lambda x.3)\ 4)\ 2 \to (\lambda x.3)\ 4 \to 3 \qquad (1)\end{aligned}$$

No α-conversion was necessary. In fact, no α-conversion is *ever* necessary with applicative-order evaluation of closed terms, because the argument substituted for a parameter in a β-reduction is closed, thus has no free variables to be captured. It is key that the term being evaluated be closed, as studied in the combinatorial weak λ-calculus [9] and closed reductions [13].

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:

$$\begin{aligned}(\lambda y.(\lambda z.\lambda y.z\ 4)\ \lambda x.y)\ 3\ 2 &\to (\lambda y.\lambda w.(\lambda x.y)\ 4)\ 3\ 2\\ &\to (\lambda w.(\lambda x.3)\ 4)\ 2 \to (\lambda x.3)\ 4 \to 3 \qquad (2)\end{aligned}$$

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 so results in the erroneous value 2:

$$\begin{aligned}(\lambda y.(\lambda z.\lambda y.z\ 4)\ \lambda x.y)\ 3\ 2 &\to (\lambda y.\lambda y.(\lambda x.y)\ 4)\ 3\ 2\\ &\to (\lambda y.(\lambda x.y)\ 4)\ 2 \to (\lambda x.2)\ 4 \to 2 \qquad (3)\end{aligned}$$

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 [26]. 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 $\sigma : \mathsf{Var} \rightharpoonup \mathsf{Irred}$ 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*

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

[1]Also known as *left-to-right call-by-value order*, 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.

Naively implemented, the rules are

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

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:

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

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 [26]. 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* [21, 26]. 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

$$\sigma : \mathsf{Var} \rightharpoonup \mathsf{Val} \qquad\qquad \mathsf{Val} = \mathsf{Const} + \mathsf{Cl}$$

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

The new reduction rules are

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

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,

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

3.4. Static Scoping with Capsules

Closures correctly capture the semantics of β-reduction with safe substitution, but at the expense of introducing extra combinatorial machinery to represent and manipulate pairs $\{\lambda x.e,\, \sigma\}$. Capsules allow us to revert to a purely λ-theoretic framework without losing the benefits of closures.

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

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

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.

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

We prove soundness formally in Section 4.

4. Soundness

In this section we show that capsule evaluation is statically scoped under applicative-order evaluation and correctly models β-reduction in the λ-calculus with safe substitution.

4.1. Evaluation Rules for Capsules

Let $d, e, \ldots$ denote λ-terms and $u, v, \ldots$ irreducible λ-terms (λ-abstractions and constants). Variables are denoted $x, y, \ldots$ and constants c, f. For any constant f denoting a function in the language, there exists an application function from terms to terms, that is also written f.

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

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

$$\langle f\ c,\, \sigma\rangle \to \langle f(c),\, \sigma\rangle \tag{5}$$

$$\langle y,\, \sigma\rangle \to \langle \sigma(y),\, \sigma\rangle \tag{6}$$

and context rules

$$\frac{\langle d,\, \sigma\rangle \xrightarrow{*} \langle d',\, \tau\rangle}{\langle d\ e,\, \sigma\rangle \xrightarrow{*} \langle d'\ e,\, \tau\rangle} \qquad \frac{\langle e,\, \sigma\rangle \xrightarrow{*} \langle e',\, \tau\rangle}{\langle v\ e,\, \sigma\rangle \xrightarrow{*} \langle v\ e',\, \tau\rangle} \tag{7}$$

where $\xrightarrow{*}$ denotes the repetition of zero or more steps of $\to$. 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 $y \in \mathsf{dom}\,\sigma$. 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

$$\langle y,\, \sigma\rangle \to \langle \sigma(y),\, \sigma\rangle \tag{8}$$

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

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

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

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

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

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

$$\langle (\lambda x.e)\ v,\, \sigma\rangle \to \langle e[x/v],\, \sigma\rangle \tag{13}$$

(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

$$(\lambda x.e)\ v \to e[x/v] \qquad f\ c \to f(c) \qquad y \to \sigma(y)$$

$$\frac{d \xrightarrow{*} d'}{(d\ e) \xrightarrow{*} (d'\ e)} \qquad \frac{e \xrightarrow{*} e'}{(v\ e) \xrightarrow{*} (v\ e')}$$

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

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

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 $\sigma : \mathsf{Var} \rightharpoonup \mathsf{Irred}$, 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 $\mathsf{FV}(e) \subseteq \{y_1, \ldots, y_k\}$ and $\mathsf{FV}(v_j) \subseteq \{y_1, \ldots, y_{j-1}\}$, $1 \leq j \leq k$, then eS^- is closed.

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

Theorem 1 $\mathsf{Eval}_\beta(e,\sigma) = \langle v, \sigma\rangle$ *if and only if there exist irreducible terms* $v_1, \ldots, v_k, u$ *and a rebinding operator* $S = [y_1/v_1]\cdots[y_k/v_k]$, *where* $y_1, \ldots, y_k$ *do not occur in* e, v, *or* σ, *such that* $\mathsf{Eval}(e,\sigma) = \langle u, \sigma S\rangle$ *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 $\mathsf{Eval}(d\ e,\sigma)$ is defined, then so is $\mathsf{Eval}_\beta(d\ e,\sigma)$, and the relationship between the two values is as described in the statement of the theorem. By definition of Eval, we have

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

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 $y_1, \ldots, y_n$ are the fresh variables and $v_1, \ldots, v_n$ 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

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

Since the variables $y_1, \ldots, y_m$ do not occur in e, they are not accessed in its evaluation, thus $\mathsf{Eval}_\beta(e,\sigma) = \langle vT^-, \sigma\rangle$. Also, since $y_{m+1}, \ldots, y_n$ do not occur in u and $y_1, \ldots, y_m$ do not occur in v, we have $uS^- = u(ST)^-$ and $vT^- = v(ST)^-$, thus

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

We thus have

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

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

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

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

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

and the implication holds in this case as well.

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

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

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

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

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

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

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

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

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

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

so the implication holds in this case as well. □

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, implemented naively as above, give 3. Capsules handle the assignment correctly, but with closures, 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. A closer look at the definitions of §3.3 shows that the definitions are mutually dependent and require a recursive, coinductive type definition [34, §11]. The types are

$$\begin{array}{ll} \mathsf{Env} = \mathsf{Var} \rightharpoonup \mathsf{Val} & \text{closure environments} \\ \mathsf{Val} = \mathsf{Const} + \mathsf{Cl} & \text{values} \\ \mathsf{Cl} = \lambda\text{-}\mathsf{Abs} \times \mathsf{Env} & \text{closures} \end{array}$$

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

A *state* is now a pair $\langle e,\, \boldsymbol{\sigma}\rangle$, where $\mathsf{FV}(e) \subseteq \mathsf{dom}\,\boldsymbol{\sigma}$, but the result of an evaluation is a Val. The evaluation semantics for closures, expressed as an interpreter Eval_c, is

$$\begin{aligned} \mathsf{Eval}_\mathrm{c}(c, \boldsymbol{\sigma}) &= c \\ \mathsf{Eval}_\mathrm{c}(\lambda x.e, \boldsymbol{\sigma}) &= \{\lambda x.e,\, \boldsymbol{\sigma}\} \\ \mathsf{Eval}_\mathrm{c}(y, \boldsymbol{\sigma}) &= \boldsymbol{\sigma}(y) \\ \mathsf{Eval}_\mathrm{c}(d\ e, \boldsymbol{\sigma}) &= \mathsf{let}\ u = \mathsf{Eval}_\mathrm{c}(d, \boldsymbol{\sigma})\ \mathsf{in} \\ &\quad\ \mathsf{let}\ v = \mathsf{Eval}_\mathrm{c}(e, \boldsymbol{\sigma})\ \mathsf{in} \\ &\quad\ \mathsf{Apply}_\mathrm{c}(u, v) \end{aligned}$$

$$\begin{aligned} \mathsf{Apply}_\mathrm{c}(f, c) &= f(c) \\ \mathsf{Apply}_\mathrm{c}(\{\lambda x.a,\, \boldsymbol{\rho}\}, v) &= \mathsf{Eval}_\mathrm{c}(a, \boldsymbol{\rho}[x/v]) \end{aligned} \tag{14}$$

The types are

$$\mathsf{Eval}_\mathrm{c} : \mathsf{Exp} \times \mathsf{Env} \rightharpoonup \mathsf{Val} \qquad \mathsf{Apply}_\mathrm{c} : \mathsf{Val} \times \mathsf{Val} \rightharpoonup \mathsf{Val}.$$

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

$$\mathsf{Apply}_{\mathrm{c}}(\{\lambda x.a,\, \boldsymbol{\rho}\}, v) = \mathsf{Eval}_{\mathrm{c}}(a[x/y], \boldsymbol{\rho}[y/v]) \quad (y \text{ fresh})$$

The corresponding large-step rules are

$$\langle c,\, \boldsymbol{\sigma}\rangle \underset{c'}{\to} c \qquad \langle \lambda x.e,\, \boldsymbol{\sigma}\rangle \underset{c'}{\to} \{\lambda x.e,\, \boldsymbol{\sigma}\} \qquad \langle y,\, \boldsymbol{\sigma}\rangle \underset{c'}{\to} \boldsymbol{\sigma}(y) \tag{15}$$

$$\frac{\langle d,\, \boldsymbol{\sigma}\rangle \overset{*}{\underset{c'}{\to}} f \qquad \langle e,\, \boldsymbol{\sigma}\rangle \overset{*}{\underset{c'}{\to}} c}{\langle d\ e,\, \boldsymbol{\sigma}\rangle \overset{*}{\underset{c'}{\to}} f(c)} \tag{16}$$

$$\frac{\langle d,\, \boldsymbol{\sigma}\rangle \overset{*}{\underset{c'}{\to}} \{\lambda x.a,\, \boldsymbol{\rho}\} \qquad \langle e,\, \boldsymbol{\sigma}\rangle \overset{*}{\underset{c'}{\to}} v \qquad \langle a[x/y],\, \boldsymbol{\rho}[y/v]\rangle \overset{*}{\underset{c'}{\to}} u}{\langle d\ e,\, \boldsymbol{\sigma}\rangle \overset{*}{\underset{c'}{\to}} u}\ (y \text{ fresh}) \tag{17}$$

The closure-converted form of a capsule $\langle e,\, \sigma\rangle$ is $\langle e,\, \overline{\sigma}\rangle$, where for any σ, we define $\overline{\sigma}$ as a map with $\mathsf{dom}\,\sigma = \mathsf{dom}\,\overline{\sigma}$ and

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

This definition is not circular, it is a well-defined coinductive definition. A thorough explanation of coinductive definitions and why they are well-defined, as well as similar examples of coinductive definitions, can be found in [34, §11].

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 $\mathsf{dom}\,\sigma \subseteq \mathsf{dom}\,\tau$ and for all $y \in \mathsf{dom}\,\sigma$, $\sigma(y) = \tau(y)$. The definition for values and closure environments is by mutual coinduction: $\sqsubseteq$ is defined to be the largest relation such that

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

Lemma 2 *The relation $\sqsubseteq$ is transitive.*

Proof. This is obvious for capsule environments.

For closure environments and values, we proceed by coinduction. Suppose $\boldsymbol{\sigma} \sqsubseteq \boldsymbol{\tau} \sqsubseteq \boldsymbol{\rho}$. Then $\mathsf{dom}\,\boldsymbol{\sigma} \subseteq \mathsf{dom}\,\boldsymbol{\tau} \subseteq \mathsf{dom}\,\boldsymbol{\rho}$, so $\mathsf{dom}\,\boldsymbol{\sigma} \subseteq \mathsf{dom}\,\boldsymbol{\rho}$, and for all $y \in \mathsf{dom}\,\boldsymbol{\sigma}$, $\boldsymbol{\sigma}(y) \sqsubseteq \boldsymbol{\tau}(y) \sqsubseteq \boldsymbol{\rho}(y)$, therefore $\boldsymbol{\sigma}(y) \sqsubseteq \boldsymbol{\rho}(y)$ 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 $u = \{\lambda x.e,\, \boldsymbol{\sigma}\}$, then $v = \{\lambda x.e,\, \boldsymbol{\tau}\}$ and $w = \{\lambda x.e,\, \boldsymbol{\rho}\}$ and $\boldsymbol{\sigma} \sqsubseteq \boldsymbol{\tau} \sqsubseteq \boldsymbol{\rho}$, therefore $\boldsymbol{\sigma} \sqsubseteq \boldsymbol{\rho}$ by the transitivity of $\sqsubseteq$ on closure environments. □

Lemma 3 *Closure conversion is monotone with respect to* $\sqsubseteq$. *That is, if* $\sigma \sqsubseteq \tau$, *then* $\overline{\sigma} \sqsubseteq \overline{\tau}$.

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

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

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

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

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

Lemma 4 $\overline{\sigma}(y) = V(\sigma(y), \sigma)$.

Proof.

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

□

Lemma 5 *If* $y \notin \mathsf{dom}\,\sigma$, *then* $\overline{\sigma}[y/V(v,\sigma)] \sqsubseteq \overline{\sigma[y/v]}$.

Proof. By Lemma 4,

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

If $y \notin \mathsf{dom}\,\sigma$, 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 3 and the last equation by (19). □

Lemma 6 *If* $\boldsymbol{\sigma} \sqsubseteq \boldsymbol{\tau}$, *then* $\mathsf{Eval_c}(e, \boldsymbol{\sigma})$ *exists if and only if* $\mathsf{Eval_c}(e, \boldsymbol{\tau})$ *does, and* $\mathsf{Eval_c}(e, \boldsymbol{\sigma}) \sqsubseteq \mathsf{Eval_c}(e, \boldsymbol{\tau})$. *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

$$\mathsf{Eval_c}(c, \boldsymbol{\sigma}) = c = \mathsf{Eval_c}(c, \boldsymbol{\tau})$$
$$\mathsf{Eval_c}(\lambda x.a, \boldsymbol{\sigma}) = \{\lambda x.a,\, \boldsymbol{\sigma}\} \sqsubseteq \{\lambda x.a,\, \boldsymbol{\tau}\} = \mathsf{Eval_c}(\lambda x.a, \boldsymbol{\tau})$$
$$\mathsf{Eval_c}(y, \boldsymbol{\sigma}) = \boldsymbol{\sigma}(y) \sqsubseteq \boldsymbol{\tau}(y) = \mathsf{Eval_c}(y, \boldsymbol{\tau}).$$

For the rule (16), $\langle d\ e,\, \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} f(c)$ is derivable by an application of (16) iff $\langle d,\, \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} f$ and $\langle e,\, \boldsymbol{\sigma}\rangle \xrightarrow[c]{*} c$ are derivable by smaller proofs. Similarly, $\langle d\ e,\, \boldsymbol{\tau}\rangle \xrightarrow[c]{*} f(c)$ is derivable

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

Finally, for the rule (17), $\langle d\ e, \boldsymbol{\sigma}\rangle \xrightarrow[c']{*} u_1$ is derivable by an application of (17) iff $\langle d, \boldsymbol{\sigma}\rangle \xrightarrow[c']{*} \{\lambda x.a, \boldsymbol{\rho}_1\}$, $\langle e, \boldsymbol{\sigma}\rangle \xrightarrow[c']{*} v_1$, and $\langle a[x/y], \boldsymbol{\rho}_1[y/v_1]\rangle \xrightarrow[c']{*} u_1$ are derivable by smaller proofs. Similarly, $\langle d\ e, \boldsymbol{\tau}\rangle \xrightarrow[c']{*} u_2$ is derivable by an application of (17) iff $\langle d, \boldsymbol{\tau}\rangle \xrightarrow[c']{*} \{\lambda x.a, \boldsymbol{\rho}_2\}$, $\langle e, \boldsymbol{\tau}\rangle \xrightarrow[c']{*} v_2$, and $\langle a[x/y], \boldsymbol{\rho}_2[y/v_2]\rangle \xrightarrow[c']{*} u_2$ are derivable by smaller proofs. By the induction hypothesis, $\langle d, \boldsymbol{\sigma}\rangle \xrightarrow[c']{*} \{\lambda x.a, \boldsymbol{\rho}_1\}$ and $\langle d, \boldsymbol{\tau}\rangle \xrightarrow[c']{*} \{\lambda x.a, \boldsymbol{\rho}_2\}$ are derivable by the same proof, and $\boldsymbol{\rho}_1 \sqsubseteq \boldsymbol{\rho}_2$. Similarly, $\langle e, \boldsymbol{\sigma}\rangle \xrightarrow[c']{*} v_1$ and $\langle e, \boldsymbol{\tau}\rangle \xrightarrow[c']{*} v_2$ are derivable by the same proof, and $v_1 \sqsubseteq v_2$. It follows that $\boldsymbol{\rho}_1[y/v_1] \sqsubseteq \boldsymbol{\rho}_2[y/v_2]$. Again by the induction hypothesis, $\langle a[x/y], \boldsymbol{\rho}_1[y/v_1]\rangle \xrightarrow[c']{*} u_1$ and $\langle a[x/y], \boldsymbol{\rho}_2[y/v_2]\rangle \xrightarrow[c']{*} u_2$ are derivable by the same proof, and $u_1 \sqsubseteq u_2$. □

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

Theorem 7 $\mathsf{Eval}(e, \sigma)$ *exists if and only if* $\mathsf{Eval}_{\mathrm{c}}(e, \overline{\sigma})$ *does, and* $\mathsf{Eval}_{\mathrm{c}}(e, \overline{\sigma}) \sqsubseteq V(\mathsf{Eval}(e, \sigma))$. *Moreover, they are derivable by isomorphic large-step proofs under the obvious correspondence between the large-step rules of both systems.*[2]

Proof. We proceed by induction on the proof tree under the large-step rules. The proof is similar to the proof of Lemma 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 $\xrightarrow{*}$.

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

$$\mathsf{Eval}_{\mathrm{c}}(c, \overline{\sigma}) = c = V(\mathsf{Eval}(c, \sigma))$$
$$\mathsf{Eval}_{\mathrm{c}}(\lambda x.a, \overline{\sigma}) = \{\lambda x.a, \overline{\sigma}\} = V(\lambda x.a, \sigma) = V(\mathsf{Eval}(\lambda x.a, \sigma))$$
$$\mathsf{Eval}_{\mathrm{c}}(y, \overline{\sigma}) = \overline{\sigma}(y) = V(\sigma(y), \sigma) = V(\mathsf{Eval}(y, \sigma)).$$

The last line uses Lemma 4.

Consider the corresponding rules (9) and (16). A conclusion $\langle d\ e, \overline{\sigma}\rangle \xrightarrow[c']{*} f(c)$ is derivable by an application of (16) iff $\langle d, \overline{\sigma}\rangle \xrightarrow[c']{*} f$ and $\langle e, \overline{\sigma}\rangle \xrightarrow[c']{*} c$ 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, $\langle d, \overline{\sigma}\rangle \xrightarrow[c']{*} f = V(f, \sigma S)$ and $\langle d, \sigma\rangle \xrightarrow{*} \langle f, \sigma S\rangle$ are derivable by isomorphic proofs. By Lemma 6, $\langle e, \overline{\sigma}\rangle \xrightarrow[c']{*} c$ and $\langle e, \overline{\sigma S}\rangle \xrightarrow[c']{*} c$ are derivable by the same proof. Again by the induction hypothesis, $\langle e, \overline{\sigma S}\rangle \xrightarrow[c']{*} c$ and $\langle e, \sigma S\rangle \xrightarrow{*} \langle c, \sigma ST\rangle$ are derivable by isomorphic proofs, therefore so are $\langle e, \overline{\sigma}\rangle \xrightarrow[c']{*} c = V(c, \sigma ST)$ and $\langle e, \sigma S\rangle \xrightarrow{*} \langle c, \sigma ST\rangle$.

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

$$\langle d, \overline{\sigma}\rangle \xrightarrow[c']{*} \{\lambda x.a, \boldsymbol{\rho}\} \qquad \langle e, \overline{\sigma}\rangle \xrightarrow[c']{*} v \qquad \langle a[x/y], \boldsymbol{\rho}[y/v]\rangle \xrightarrow[c']{*} u$$

[2] For 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).

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,

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

are derivable by smaller proofs.

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

By Lemmas 3 and 6, for some v', $\langle e,\ \overline{\sigma}\rangle \xrightarrow[c]{*} v$ and $\langle e,\ \overline{\sigma S}\rangle \xrightarrow[c]{*} v'$ are derivable by the same proof, and $v \sqsubseteq v'$. Again by the induction hypothesis, $\langle e,\ \overline{\sigma S}\rangle \xrightarrow[c]{*} v'$ 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, $\langle e,\ \overline{\sigma}\rangle \xrightarrow[c]{*} v$ 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 5,

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

Again by Lemma 6, for some u', $\langle a[x/y],\ \boldsymbol{\rho}[y/v]\rangle \xrightarrow[c]{*} u$ and $\langle a[x/y],\ \overline{\sigma ST[y/w]}\rangle \xrightarrow[c]{*} u'$ are derivable by the same proof, and $u \sqsubseteq u'$; and again by the induction hypothesis, $\langle a[x/y],\ \overline{\sigma ST[y/w]}\rangle \xrightarrow[c]{*} u'$ 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, $\langle a[x/y],\ \boldsymbol{\rho}[y/v]\rangle \xrightarrow[c]{*} u$ and $\langle a[x/y],\ \sigma ST[y/w]\rangle \xrightarrow[c]{*} \langle t,\ \tau\rangle$ are derivable by isomorphic proofs, and $u \sqsubseteq V(t, \tau)$. □

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. We thus fulfill the original desire to provide a unified semantics for functional and imperative languages.

5.1. Expressions

Expressions $\mathsf{Exp} = \{d, e, \ldots\}$ contain both functional and imperative features. There is an unlimited supply of *variables* $x, y, \ldots$ of all (simple) types, as well as constants $f, c, \ldots$ 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 $\mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e$
- repeat loop $\mathsf{repeat}\ e\ \mathsf{until}\ b$,

and syntactic sugar

- $\mathsf{let}\ x = d\ \mathsf{in}\ e$ $(\lambda x.e)\ d$

- $\mathsf{let\ rec}\ f = g\ \mathsf{in}\ e \qquad \mathsf{let}\ f = h\ \mathsf{in}\ f := g; e$

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 $\to$ representing *partial* functions. Every variable carries its own (unique) type. The typing rules are:

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

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

5.3. Evaluation

A *value* v is an expression that is either a λ-abstraction or a constant. A *capsule value* is the equivalence class of an irreducible capsule modulo bisimilarity and α-conversion; equivalently, the λ-coterm represented by the capsule modulo α-conversion. It is also a capsule whose first element is a value.

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:

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

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

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

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

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

$$\frac{\langle e,\, \sigma\rangle \xrightarrow{*} \langle v,\, \rho\rangle \qquad \langle b,\, \rho\rangle \xrightarrow{*} \langle \mathsf{true},\, \tau\rangle}{\langle \mathsf{repeat}\ e\ \mathsf{until}\ b,\, \sigma\rangle \xrightarrow{*} \langle v,\, \tau\rangle}$$

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

5.4. Garbage Collection

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

- $\tau(h(x)) = h(\sigma(x))$ for all $x \in \mathsf{dom}\,\sigma$, 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

Capsules provide an algebraic representation of state for higher-order functional and imperative programs. They are mathematically simpler than closures and correctly model static 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.

Subsequent to this work, the relationship between capsules and closures established in Theorem 7 has been strengthened to small-step bisimulation [16]. Also, with appropriate extensions to the definition of closure to allow indirection, the same relationship has been shown to hold in the presence of assignment [15]. Capsules have also been used to model objects [19] and to provide a semantics for separation logic [17].

Acknowledgments

Thanks to Robert Constable, Matthias Felleisen, Nate Foster, Konstantinos Mamouras, Andrew Myers, Mark Reitblatt, Fred Schneider, Alexandra Silva, Ross Tate, and all the members of the PLDG seminar at Cornell for valuable discussions.

References

[1] M. Abadi, L. Cardelli, *A Theory of Objects*. Springer, 1996.

[2] M. Abadi, L. Cardelli, P.-L. Curien, J.-J. Lévy, Explicit substitutions. *Journal of Functional Programming* **1** (1991) 4, 375–416.

[3] K. Aboul-Hosn, Programming With Private State. Honors Thesis, The Pennsylvania State University, 2001.
`http://www.cs.cornell.edu/%7Ekamal/thesis.pdf`

[4] K. Aboul-Hosn, D. Kozen, Relational semantics for higher-order programs. In: T. Uustalu (ed.), *Proc. 8th Int. Conf. Mathematics of program construction (MPC'06)*. LNCS 4014, Springer-Verlag, 2006, 29–48.

[5] K. ABOUL-HOSN, D. KOZEN, Local variable scoping and Kleene algebra with tests. *Journal of Logic and Algebraic Programming* **76** (2008) 1, 3–17. DOI: 10.1016/j.jlap.2007.10.007.

[6] S. ABRAMSKY, K. HONDA, 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*. IEEE Computer Society, Washington, DC, USA, 1998, 334–344.

[7] S. ABRAMSKY, G. MCCUSKER, Linearity, sharing and state: a fully abstract game semantics for idealized ALGOL with active expressions. *Electronical Notes in Theoretical Computer Science* **3** (1996).

[8] H. P. BARENDREGT, J. W. KLOP, Applications of infinitary lambda calculus. *Information and Computation* **207** (2009) 5, 559–582.

[9] N. ÇAGMAN, J. R. HINDLEY, Combinatory weak reduction in lambda calculus. *Theoretical Computer Science* **198** (1998) 1-2, 239–247.

[10] P.-L. CURIEN, An abstract framework for environment machines. *Theoretical Computer Science* **82** (1991) 2, 389–402.

[11] M. FELLEISEN, R. B. FINDLER, M. FLATT, *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[12] M. FELLEISEN, R. HIEB, The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science* **103** (1992), 235–271.

[13] M. FERNÁNDEZ, I. MACKIE, F.-R. SINOT, Closed reduction: explicit substitutions without alpha-conversion. *Mathematical Structures in Computer Science* **15** (2005) 2, 343–381.

[14] J. Y. HALPERN, A. R. MEYER, 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, 245–257.

[15] J.-B. JEANNIN, Capsules and closures. *Electronical Notes in Theoretical Computer Science* **276** (2011), 191–213. `http://dx.doi.org/10.1016/j.entcs.2011.09.022`

[16] J.-B. JEANNIN, Capsules and closures: a small-step approach. In: R. L. CONSTABLE, A. SILVA (eds.), *Kozen Festschrift*. LNCS 7230, Springer-Verlag, 2012, 106–123.

[17] J.-B. JEANNIN, D. KOZEN, Capsules and separation. In: N. DERSHOWITZ (ed.), *Proc. 27th ACM/IEEE Symp. Logic in Computer Science (LICS'12)*. IEEE, Dubrovnik, Croatia, 2012, 425–430.

[18] J. W. KLOP, R. C. DE VRIJER, Infinitary normalization. In: S. ARTEMOV, H. BARRINGER, A. S. D'AVILA GARCEZ, L. C. LAMB, J. WOODS (eds.), *We Will Show Them: Essays in Honour of Dov Gabbay*. 2, College Publications, 2005, 169–192.

[19] D. Kozen, New. In: U. Berger, M. Mislove (eds.), *Proc. 28th Conf. Math. Found. Programming Semantics (MFPS XXVIII)*. Elsevier Electronic Notes in Theoretical Computer Science, Bath, England, 2012, 13–38.

[20] J. Laird, A game semantics of local names and good variables. In: I. Walukiewicz (ed.), *FoSSaCS*. LNCS 2987, Springer-Verlag, 2004, 289–303.

[21] P. J. Landin, The mechanical evaluation of expressions. *Computer Journal* **6** (1964) 4, 308–320.

[22] I. Mason, C. Talcott, Programming, transforming, and proving with function abstractions and memories. In: G. Ausiello, M. Dezani-Ciancaglini, S. Ronchi della Rocca (eds.), *Automata, Languages, and Programming*. LNCS 372, Springer-Verlag, 1989, 574–588.

[23] I. Mason, C. Talcott, Axiomatizing operational equivalence in the presence of side effects. In: *Fourth Annual Symposium on Logic in Computer Science. IEEE*. IEEE Computer Society Press, 1989, 284–293.

[24] I. Mason, C. Talcott, Equivalence in functional languages with effects. *Journal of Functional Programming* **1** (1991) 3, 287–327.

[25] I. A. Mason, C. L. Talcott, References, local variables and operational reasoning. In: *Seventh Annual Symposium on Logic in Computer Science*. IEEE, 1992, 186–197.
`http://www-formal.stanford.edu/MT/92lics.ps.Z`

[26] J. McCarthy, History of LISP. In: R. L. Wexelblat (ed.), *History of programming languages I*. ACM, 1981, 173–185.

[27] R. Milne, C. Strachey, *A Theory of Programming Language Semantics*. Halsted Press, New York, NY, USA, 1977.

[28] E. Moggi, Notions of computation and monads. *Information and Computation* **93** (1991) 1, 55–92.

[29] A. Moran, D. Sands, Improvement in a lazy context: an operational theory for call-by-need. In: *POPL*. 1999, 43–56.

[30] A. M. Pitts, Operationally-based theories of program equivalence. In: P. Dybjer, A. M. Pitts (eds.), *Semantics and Logics of Computation*. Publications of the Newton Institute, Cambridge University Press, 1997, 241–298.
`http://www.cs.tau.ac.il/~nachumd/formal/exam/pitts.pdf`

[31] A. M. Pitts, *Operational Semantics and Program Equivalence*. Technical report, INRIA Sophia Antipolis, 2000. Lectures at the International Summer School On Applied Semantics, APPSEM 2000, Caminha, Minho, Portugal, September 2000.
`http://www.springerlink.com/media/1f99vvygyh3ygrykklby/contributions/l/w/f/6/lwf6r3jxn7a2lkq0.pdf`

[32] A. M. Pitts, I. D. B. Stark, Observable properties of higher order functions that dynamically create local names, or What's new? In: A. M. Borzyszkowski, S. Sokolowski (eds.), *Mathematical Foundations of Computer Science*. LNCS 711, Springer-Verlag, 1993, 122–141.

[33] A. M. PITTS, I. D. B. STARK, Operational reasoning in functions with local state. In: A. D. GORDON, A. M. PITTS (eds.), *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1998, 227–273. `http://homepages.inf.ed.ac.uk/stark/operfl.pdf`

[34] J. RUTTEN, Universal coalgebra: a theory of systems. *Theoretical Computer Science* **249** (2000), 3–80.

[35] D. SCOTT, Mathematical concepts in programmng language semantics. In: *Proc. 1972 Spring Joint Computer Conferences*. AFIPS Press, Montvale, NJ, 1972, 225–34.

[36] J. E. STOY, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1981.

(Received: November 21, 2012; revised: May 13, 2013)