Capsules and Closures: a Small-Step Approach

Jean-Baptiste Jeannin

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501, USA
jeannin@cs.cornell.edu

Abstract. We present a side by side comparison of *Capsules* and *Closures*, including a proof of bisimilarity, using small-step semantics. A similar proof was presented in [8], using big-step semantics. However, while big-step semantics only allow to talk about final results of terminating computations, the use of small-step semantics allows to prove a stronger bisimilarity involving every step of the computation and thus also applicable to infinite computations.

1 Introduction

This paper compares the use of Capsules and Closures for a language with both higher-order and imperative features, and using small-step semantics. Capsules, introduced in [9], are a simple way of modeling the state of a computation for languages that are both imperative and functional. The state of a computation has been studied extensively [1,2,6,7,12–16,20,21]. However capsules intend to be as simple as possible, using only a single environment, while still capturing lexical scoping, variable assignment and recursion without heaps, stacks or combinators, and only using simple types.

The first versions of Lisp implemented *dynamic scoping*, which did not follow the semantics of the λ-calculus based on β-reduction. The language Scheme [22] fixed this by introducing *closures*, which allow to correctly model static scoping. The idea behind closures is to keep with each λ–abstraction the environment in which it was declared, thus forming a closure and allowing to execute the body of the λ-abstraction in the correct environment.

The language we introduce is both functional and imperative: it has higher-order functions, but every variable is mutable. This leads to interesting interactions and allows to go further than just enforcing lexical scoping. In particular, what do we expect the result of an expression like ($\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f0$) to be? Scheme (using $\mathsf{set!}$ for $:=$) and OCaml (using references) answer 2. Capsules give a rigorous mathematical definition that agrees and conservatively extends the scoping rules of the λ-calculus. Our semantics of closures also agrees with this definition, but this requires introducing a level of indirection, with both a stack of environments and a store, à la ML. Finally, recursive definitions are

often implemented using some sort of backpatching; we build this directly into the definition of the language by defining $\mathsf{let\ rec}\ x = d\ \mathsf{in}\ e$ as a syntactic sugar for $\mathsf{let}\ x = a\ \mathsf{in}\ x := d; e$, where a is any expression of the appropriate type.

There is much previous work on reasoning about references and local state; see [11, 17–19]. State is typically modeled by some form of heap from which storage locations can be allocated and deallocated [7,15,20,21]. Others have used game semantics to reason about local state [4, 5, 10]. Mason and Talcott [12–14] and Felleisen and Hieb [6] present a semantics based on a heap and storage locations. A key difference is that Felleisen and Hieb's semantics is based on continuations. Finally, Moggi [16] proposed monads, which can be used to model state and are implemented in Haskell.

This paper is an improvement over [8] and borrows most of its structure, as we prove the same kind of result. Here we use small-step semantics instead of big-step semantics, which allows to prove bisimilar even infinite computations. The language and the notations are the same. The intended semantics of the language, both using capsules and closures, is the same as well, even though we have not formally proven the big-step and the small-step versions equivalent.

This paper is organized as follows. In Sect. 2, we formally introduce a programming language based on the λ-calculus containing both functional and imperative features. In Sect. 3, we describe two semantics for this language, one based on capsules and the other on closures. In Sect. 4, we show a very strong correspondence (Theorems 1 and 2, Corollary 1) between the two semantics, showing that every computation in the semantics of capsules is bisimilar to a computation in the semantics of closures, and vice-versa. We finish with a discussion in Sect. 5.

2 Syntax

2.1 Expressions

Expressions $\mathsf{Exp} = \{d, e, a, b, \ldots\}$ contain both functional and imperative features. There is an unlimited supply of *variables* $x, y, z, \ldots$ of all (simple) types, as well as *constants* $f, c, \ldots$ for primitive values. () is the only constant of type unit, and true and false are the only two constants of type bool. In the examples, $0, 1, 2, \ldots$ are predefined constants of type int. In addition, there are functional features

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

imperative features

- assignment $\quad x := e$

– composition $d; e$

– conditional $\mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e$

– while loop $\mathsf{while}\ b\ \mathsf{do}\ e$,

and syntactic sugars

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

– $\mathsf{let\,rec}\ x = d\ \mathsf{in}\ e$ $\mathsf{let}\ x = a\ \mathsf{in}\ x := d; e$

where a is any expression of the appropriate type.

Let Var be the set of variables, Const the set of constants, and λ-Abs the set of λ-abstractions. Given an expression e, let $\mathsf{FV}(e)$ denote the set of free variables of e. Given a partial function $h : \mathsf{Var} \rightharpoonup \mathsf{Var}$ such that $\mathsf{FV}(e) \subseteq \mathsf{dom}\, h$, let $h(e)$ be the expression e where every instance of a free variable $x \in \mathsf{FV}(e)$ has been replaced by the variable $h(x)$. As usual, given two partial functions g and h, $g \circ h$ denotes their composition such that for all x, $g \circ h(x) = g(h(x))$. Given a function h, we write $h[x/v]$ the function such that $h[x/v](y) = h(y)$ for $y \neq x$ and $h[x/v](x) = v$. Given an expression e, we write $e[x/y]$ the expression e where all free occurrences of x have been replaced by y.

Throughout the paper, we focus on the features directly involving variables: variable calls x, λ-abstractions $\lambda x.e$, applications $(d\ e)$ where d reduces to a λ-abstraction, and assignment $x := e$. Most differences between capsules and closures arise using these features.

2.2 Types

Types $\alpha, \beta, \ldots$ are built inductively from an unspecified family of base types, including at least unit and bool, and a type constructor $\to$ such that functions with input type α and return type β have type $\alpha \to \beta$. All constants c of the language have a type $\mathsf{type}(c)$; by convention, we use c for a constant of a base type and f for a constant of a functional type. We follow [23] in assuming that each variable x is associated with a unique type $\mathsf{type}(x)$, that could for example be built into the variable name. Γ is a type environment, a partial function $\mathsf{Var} \rightharpoonup \mathsf{Type}$. As is standard, we write $\Gamma, x : \alpha$ for the typing environment Γ where x has been bound or rebound to α. The typing rules are standard:

$$\Gamma \vdash c : \alpha \text{ if } \mathsf{type}(c) = \alpha \qquad \Gamma, x : \alpha \vdash x : \alpha \qquad \frac{\mathsf{type}(x) = \alpha \quad \Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha \to \beta}$$

$$\frac{\Gamma \vdash d : \alpha \to \beta \quad \Gamma \vdash e : \alpha}{\Gamma \vdash (d\ e) : \beta} \qquad \frac{\Gamma \vdash x : \alpha \quad \Gamma \vdash e : \alpha}{\Gamma \vdash x := e : \mathsf{unit}} \qquad \frac{\Gamma \vdash d : \mathsf{unit} \quad \Gamma \vdash e : \alpha}{\Gamma \vdash d; e : \alpha}$$

$$\frac{\Gamma \vdash b : \mathsf{bool} \quad \Gamma \vdash d : \alpha \quad \Gamma \vdash e : \alpha}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e : \alpha} \qquad \frac{\Gamma \vdash b : \mathsf{bool} \quad \Gamma \vdash e : \mathsf{unit}}{\Gamma \vdash \mathsf{while}\ b\ \mathsf{do}\ e : \mathsf{unit}}$$

3 Semantics

We present two different semantics that have a strong correspondence:

- The semantics on *capsules* is a simplified version of the semantics on closure structures introduced in [3]. It has previously been described in [9];
- The semantics on *closures* is the semantics usually used and taught for functional languages. A level of indirection for variables has been added to support imperative features, *à la* ML. Developing a small-step semantics for closures ended up being a little bit more complicated than expected.

From now on all the expressions we consider are supposed well-typed with the rules of Sect. 2.2.

3.1 Capsules

Definitions An *irreducible term* is either a constant or a λ-abstraction. A *capsule environment* γ is a partial function from variables to irreducible terms, such that

$$\forall x \in \mathsf{dom}\,\gamma,\ \mathsf{FV}(\gamma(x)) \subseteq \mathsf{dom}\,\gamma$$

.

Let $i, j, k, \ldots$ denote irreducible terms and $\gamma, \delta, \zeta, \eta, \ldots$ capsule environments. Let $\mathsf{Irred} = \mathsf{Const} + \lambda\text{-}\mathsf{Abs}$ be the set of irreducible terms. Thus we have:

$$\gamma : \mathsf{Var} \rightharpoonup \mathsf{Irred} \qquad\qquad \mathsf{Irred} = \mathsf{Const} + \lambda\text{-}\mathsf{Abs}$$

Semantics A *capsule* is a pair $\langle e, \gamma \rangle$, such that $\mathsf{FV}(e) \subseteq \mathsf{dom}\,\gamma$.

Let us define a small step semantics where the operator $\rightarrow_{\mathsf{ca}}$ relates capsules. The semantics of features directly involving variables is given by:

$$\langle x, \gamma \rangle \rightarrow_{\mathsf{ca}} \langle \gamma(x), \gamma \rangle \qquad \langle y := i, \gamma \rangle \rightarrow_{\mathsf{ca}} \langle (), \gamma[y/i] \rangle$$
$$\langle (\lambda x.b)\ i, \gamma \rangle \rightarrow_{\mathsf{ca}} \langle b[x/y], \gamma[y/i] \rangle \quad (y \text{ fresh})$$

and the remaining semantics is:

$$\langle f\ c, \gamma \rangle \rightarrow_{\mathsf{ca}} \langle f(c), \gamma \rangle \qquad \langle (); e, \gamma \rangle \rightarrow_{\mathsf{ca}} \langle e, \gamma \rangle$$
$$\langle \mathsf{if\ true\ then}\ d\ \mathsf{else}\ e, \gamma \rangle \rightarrow_{\mathsf{ca}} \langle d, \gamma \rangle \qquad \langle \mathsf{if\ false\ then}\ d\ \mathsf{else}\ e, \gamma \rangle \rightarrow_{\mathsf{ca}} \langle e, \gamma \rangle$$
$$\langle \mathsf{while}\ b\ \mathsf{do}\ e, \gamma \rangle \rightarrow_{\mathsf{ca}} \langle \mathsf{if}\ b\ \mathsf{then}\ (e; \mathsf{while}\ b\ \mathsf{do}\ e)\ \mathsf{else}\ (), \gamma \rangle$$

Evaluation contexts C are defined by:

$$C ::= \quad [\cdot] \quad | \quad C\ e \quad | \quad i\ C \quad | \quad x := C \quad | \quad C; e \quad | \quad \mathsf{if}\ C\ \mathsf{then}\ d\ \mathsf{else}\ e$$

where each evaluation context $C[\,\cdot\,]$ generates a rule:

$$\frac{\langle d,\, \gamma\rangle \to_{\mathsf{ca}} \langle e,\, \delta\rangle}{\langle C[d],\, \gamma\rangle \to_{\mathsf{ca}} \langle C[e],\, \delta\rangle}$$

The well-typed final capsules, i.e. capsules that cannot take a small step, are exactly the capsules $\langle i,\, \gamma\rangle$ for any irreducible term i.

As usual, we introduce $\to_{\mathsf{ca}}^*$ as the reflexive transitive closure of $\to_{\mathsf{ca}}$.

Examples The following examples show that lexical scoping and recursion are handled.

Example 1. $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0) \to_{\mathsf{ca}}^* 1$

Proof.

$$\begin{array}{rlr}
 & \langle\ \mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0, & [\,]\ \rangle \\
\to_{\mathsf{ca}} & \langle\ \mathsf{let}\ f = \lambda y.x'\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0, & [x' = 1]\ \rangle \\
\to_{\mathsf{ca}} & \langle\ \mathsf{let}\ x = 2\ \mathsf{in}\ f'\ 0, & [x' = 1,\ f' = \lambda y.x']\ \rangle \\
\to_{\mathsf{ca}} & \langle\ f'\ 0, & [x' = 1,\ f' = \lambda y.x',\ x'' = 2]\ \rangle \\
\to_{\mathsf{ca}} & \langle\ (\lambda y.x')\ 0, & [x' = 1,\ f' = \lambda y.x',\ x'' = 2]\ \rangle \\
\to_{\mathsf{ca}} & \langle\ x', & [x' = 1,\ f' = \lambda y.x',\ x'' = 2,\ y' = 0]\ \rangle \\
\to_{\mathsf{ca}} & \langle\ 1, & [x' = 1,\ f' = \lambda y.x',\ x'' = 2,\ y' = 0]\ \rangle
\end{array}$$

Example 2. $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0) \to_{\mathsf{ca}}^* 2$

Proof.

$$\begin{array}{rlr}
 & \langle\ \mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0, & [\,]\ \rangle \\
\to_{\mathsf{ca}} & \langle\ \mathsf{let}\ f = \lambda y.x'\ \mathsf{in}\ x' := 2; f\ 0, & [x' = 1]\ \rangle \\
\to_{\mathsf{ca}} & \langle\ x' := 2; f'\ 0, & [x' = 1,\ f' = \lambda y.x']\ \rangle \\
\to_{\mathsf{ca}}^* & \langle\ f'\ 0, & [x' = 2,\ f' = \lambda y.x']\ \rangle \\
\to_{\mathsf{ca}} & \langle\ (\lambda y.x')\ 0, & [x' = 2,\ f' = \lambda y.x']\ \rangle \\
\to_{\mathsf{ca}} & \langle\ x', & [x' = 2,\ f' = \lambda y.x',\ y' = 0]\ \rangle \\
\to_{\mathsf{ca}} & \langle\ 2, & [x' = 2,\ f' = \lambda y.x',\ y' = 0]\ \rangle
\end{array}$$

Example 3. $(\mathsf{let\,rec}\ f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n\ \mathsf{in}\ f\ 3) \to_{\mathsf{ca}}^* 6$

Proof. In this example e stands for $\lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n$.

$$
\begin{array}{lll}
 & \langle\ \mathsf{let\,rec}\ f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n\ \mathsf{in}\ f\ 3, & [\,]\ \rangle \\
\to^*_{\mathsf{ca}} & \langle\ f\ 3, & [f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n]\ \rangle \\
\to^*_{\mathsf{ca}} & \langle\ \mathsf{if}\ n_1 = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n_1 - 1) \times n_1, & [f = e,\ n_1 = 3]\ \rangle \\
\to^*_{\mathsf{ca}} & \langle\ (f\ 2) \times n_1, & [f = e,\ n_1 = 3]\ \rangle \\
\to^*_{\mathsf{ca}} & \langle\ (\mathsf{if}\ n_2 = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n_2 \times f(n_2 - 1)) \times n_1, & [f = e,\ n_1 = 3,\ n_2 = 2]\ \rangle \\
\to^*_{\mathsf{ca}} & \langle\ (f\ 1) \times n_2 \times n_1, & [f = e,\ n_1 = 3,\ n_2 = 2]\ \rangle \\
\to^*_{\mathsf{ca}} & \langle\ (\mathsf{if}\ n_3 = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n_3 \times f(n_3 - 1)) \times n_2 \times n_1, & \\
 & & [f = e,\ n_1 = 3,\ n_2 = 2,\ n_3 = 1]\ \rangle \\
\to^*_{\mathsf{ca}} & \langle\ (f\ 0) \times n_3 \times n_2 \times n_1, & [f = e,\ n_1 = 3,\ n_2 = 2,\ n_3 = 3]\ \rangle \\
\to^*_{\mathsf{ca}} & \langle\ (\mathsf{if}\ n_4 = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n_4 \times f(n_4 - 1)) \times n_3 \times n_2 \times n_1, & \\
 & & [f = e,\ n_1 = 3,\ n_2 = 2,\ n_3 = 1,\ n_4 = 0]\ \rangle \\
\to^*_{\mathsf{ca}} & \langle\ 1 \times n_3 \times n_2 \times n_1, & [f = e,\ n_1 = 3,\ n_2 = 2,\ n_3 = 1,\ n_4 = 0]\ \rangle \\
\to^*_{\mathsf{ca}} & \langle\ 6, & [f = e,\ n_1 = 3,\ n_2 = 2,\ n_3 = 1,\ n_4 = 0]\ \rangle
\end{array}
$$

3.2 Closures

Definitions Closures were introduced in the language Scheme [22]. We present a version of them using a level of indirection, allowing us to handle mutable variables.

There is an unlimited number of locations $\ell, \ell_1, \ell_2 \ldots$; locations can be thought of as addresses in memory. An *environment* is a partial function from variables to locations. A *closure* is defined as a pair $\{\lambda x.e,\ \sigma\}$ such that $\mathsf{FV}(\lambda x.e) \subseteq \mathsf{dom}\,\sigma$, where $\lambda x.e$ is a λ-abstraction and σ is an environment that is used to interpret the free variables of $\lambda x.e$. A *value* is either a constant or a closure. Values for closures play the same role as irreducible terms for capsules. A *store* (or *memory*) is a partial function from locations to values.

Let $u, v, w, \ldots$ denote values, $\sigma, \tau, \ldots$ environments and $\mu, \nu, \xi, \chi, \ldots$ stores. Let Val be the set of values, Loc the set of locations and Cl the set of closures. Thus we have:

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

The interaction of small-step semantics and closures leads to using stacks of environments: when entering the body of a function, the environment coming with its closure is pushed; and when leaving this body, it is popped. Let $\Sigma, \Pi, \ldots$ denote stacks of environments. Let us write $[\sigma]$ the stack of environments containing only the element σ, as to not confuse it with the single environment σ. $\sigma :: \tau$ represents the stack containing σ at the top of the stack and τ at its

bottom. $\sigma :: \Sigma$ represents the stack Σ with σ added on top of it; and $\Sigma :: \sigma$ represents Σ with σ added at its bottom. We define $\mathsf{hd}(\Sigma) = \sigma$ and $\mathsf{tl}(\Sigma) = \Pi$ whenever $\Sigma = \sigma :: \Pi$.

To define a small-step semantics of closures, we need to represent all the different shapes an expression can take throughout computation, until it becomes a value. State expressions $\mathsf{StExp} = \{s, t, \ldots\}$ allow to do this. Of course, expressions and values are state expressions, but some state expressions are neither.

$$\mathsf{Exp} \subseteq \mathsf{StExp} \qquad\qquad \mathsf{Val} \subseteq \mathsf{StExp}$$

A *state expression* can be:

- an expression e; this includes constants c;
- a closure $\{\lambda x.a,\, \sigma\}$;
- an evaluation expression followed by a pop indicator $\square$, as in $s\ \square$; when entering the body of a function, a new environment needs to be pushed on the stack of environments; this environment needs to be popped when leaving the body of the function; one way to know when this happens is to keep track of the end of the body with $\square$;
- an evaluation expression applied to an expression, $s\ e$;
- a value applied to an evaluation expression, $v\ s$;
- an assignment, $x := s$;
- a composition $s; e$;
- an if statement $\mathsf{if}\ s\ \mathsf{then}\ d\ \mathsf{else}\ e$.

We extend the notion of free variables to an evaluation expression in a natural, syntactic way: for a well-formed closure $\{\lambda x.a,\, \sigma\}$ with $\mathsf{FV}(\lambda x.a) \subseteq \mathsf{dom}\,\sigma$, $\mathsf{FV}(\{\lambda x.a,\, \sigma\})$ is the empty set; and $\mathsf{FV}(s\ \square) = \mathsf{FV}(s)$.

Stacks of environments, along with the introduction of closures and of the pop indicator $\square$, are a convenient way to model in which environment each variable should be looked up. Intuitively, any free variable in a λ-abstraction of a closure should be interpreted in the environment coming with this closure. Because of the definition of state expressions, the pop indicators all are inside each other. The variables inside the deepest pop indicator are interpreted in the environment on top the stack; the variables inside the second deepest but outside the deepest pop indicator are interpreted in the second environment from the top of the stack, and so on. The variables outside of any pop indicator are interpreted in the environment at the bottom of the stack. A precise account of this idea will be given in Sect. 4.1 with the definition of $h \circ \Sigma$.

Semantics A *state* is a triple $(s,\, \Sigma,\, \mu)$.

$$\mathsf{FV}(s) \subseteq \mathsf{dom}\,(\mathsf{hd}\,\Sigma) \qquad \forall \sigma \in \Sigma, \mathsf{codom}\,\sigma \subseteq \mathsf{dom}\,\mu$$
$$\forall \{\lambda x.a,\, \tau\} \in \mathsf{codom}\,\mu,\ \mathsf{FV}(\lambda x.a) \subseteq \mathsf{dom}\,\tau \wedge \mathsf{codom}\,\tau \subseteq \mathsf{dom}\,\mu$$

When evaluating an expression e, we start with the initial state $(e,\, [\sigma],\, \mu)$ where σ and μ are both empty mappings. Let us define a small step semantics where the operator $\to_{\mathsf{cl}}$ relates valid states to valid results. The semantics of features directly involving variables is given by:

$$(x,\, [\sigma],\, \mu) \to_{\mathsf{cl}} (\mu(\sigma(x)),\, [\sigma],\, \mu) \qquad (\lambda x.a,\, [\sigma],\, \mu) \to_{\mathsf{cl}} (\{\lambda x.a,\, \sigma\},\, [\sigma],\, \mu)$$
$$(\{\lambda x.a, \sigma\}\ v,\, [\tau],\, \mu) \to_{\mathsf{cl}} (a\ \square,\, \sigma[x/\ell] :: \tau,\, \mu[\ell/v]) \quad (\ell \text{ fresh})$$
$$(v\ \square,\, \sigma :: \tau,\, \mu) \to_{\mathsf{cl}} (v,\, [\tau],\, \mu) \qquad (x := v,\, [\sigma],\, \mu) \to_{\mathsf{cl}} ((),\, [\sigma],\, \mu[\sigma(x)/v])$$

and the remaining semantics is:

$$(f\ c,\, [\sigma],\, \mu) \to_{\mathsf{cl}} (f(c),\, [\sigma],\, \mu) \qquad ((); e,\, [\sigma],\, \mu) \to_{\mathsf{cl}} (e,\, [\sigma],\, \mu)$$
$$(\mathsf{if\ true\ then}\ d\ \mathsf{else}\ e,\, [\sigma],\, \mu) \to_{\mathsf{cl}} (d,\, [\sigma],\, \mu)$$
$$(\mathsf{if\ false\ then}\ d\ \mathsf{else}\ e,\, [\sigma],\, \mu) \to_{\mathsf{cl}} (e,\, [\sigma],\, \mu)$$
$$(\mathsf{while}\ b\ \mathsf{do}\ e,\, [\sigma],\, \mu) \to_{\mathsf{cl}} (\mathsf{if}\ b\ \mathsf{then}\ (e; \mathsf{while}\ b\ \mathsf{do}\ e)\ \mathsf{else}\ (),\, [\sigma],\, \mu)$$

Evaluation contexts C are defined by:

$$C ::= \quad [\,\cdot\,] \quad | \quad C\ e \quad | \quad v\ C \quad | \quad x := C \quad | \quad C; e \quad | \quad \mathsf{if}\ C\ \mathsf{then}\ d\ \mathsf{else}\ e$$

where each evaluation context $C[\,\cdot\,]$ generates a rule:

$$\frac{(s,\, \Sigma,\, \mu) \to_{\mathsf{cl}} (t,\, \Pi,\, \nu)}{(C[s],\, \Sigma,\, \mu) \to_{\mathsf{cl}} (C[t],\, \Pi,\, \nu)}$$

One more rule is needed to be able to evaluate under a pop indicator $\square$:

$$\frac{(s,\, \Sigma,\, \mu) \to_{\mathsf{cl}} (t,\, \Pi,\, \nu)}{(s\ \square,\, \Sigma :: \sigma,\, \mu) \to_{\mathsf{cl}} (t\ \square,\, \Pi :: \sigma,\, \nu)}$$

Note the similarity between the last two rules, including the definition of evaluation contexts, and the inductive definition of state environments. This is not by chance: the innermost state expression, if not a value, is always the one which will be evaluated next.

The final states, i.e., the states that cannot take a small step, are the (v, Σ, μ) for any value v.

As usual, we introduce $\to_{\mathsf{cl}}^*$ as the reflexive transitive closure of $\to_{\mathsf{cl}}$.

Properties Some properties of this semantics can be easily proved:

- In an evaluation expression s, all the pop indicators $\square$ are inside each other; the deepest one is inside all the others, and so on.

- If starting from an initial state, the number of elements on the environment stack Σ is always one more than the number of pop indicators $\square$ in s.

- if $\Sigma \rightarrow_{\mathsf{cl}} \Pi$ then either $\Sigma = \Pi$ or $\Sigma = \sigma :: \Pi$ or $\sigma :: \Sigma = \Pi$ for some σ.

Examples To illustrate the above semantics, we now show the evaluation of the examples of section 3.1 using closures.

Example 4. $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0) \rightarrow_{\mathsf{cl}}^{*} 1$

Proof.

$$
\begin{array}{ll}
 & \langle\ \mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0, \quad [\,], [\,]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ \{\lambda x.\mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0, [\,]\}\ 1, \quad [\,], [\,]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0\ \square, \quad [x = \ell_1] :: [\,], [\ell_1 = 1]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ \{\lambda f.\mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0, [x = \ell_1]\}\ \lambda y.x\ \square, \quad [x = \ell_1] :: [\,], [\ell_1 = 1]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ \{\lambda f.\mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0, [x = \ell_1]\}\ \{\lambda y.x, [x = \ell_1]\}\ \square, \quad [x = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = 1]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ \mathsf{let}\ x = 2\ \mathsf{in}\ f\ 0\ \square\ \square, \quad [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = 1, \ell_2 = \{\lambda y.x, [x = \ell_1]\}]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ \{\lambda x.f\ 0, [x = \ell_1, f = \ell_2]\}\ 2\ \square\ \square, \quad [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = 1, \ell_2 = \{\lambda y.x, [x = \ell_1]\}]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ f\ 0\ \square\ \square\ \square, \quad [f = \ell_2, x = \ell_3] :: [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = 1, \ell_2 = \{\lambda y.x, [x = \ell_1]\}, \ell_3 = 2]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ \{\lambda y.x, [x = \ell_1]\}\ 0\ \square\ \square\ \square, \\
 & \qquad [f = \ell_2, x = \ell_3] :: [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = 1, \ell_2 = \{\lambda y.x, [x = \ell_1]\}, \ell_3 = 2]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ x\ \square\ \square\ \square\ \square, \\
 & \quad [x = \ell_1, y = \ell_4] :: [f = \ell_2, x = \ell_3] :: [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = 1, \ell_2 = \{\lambda y.x, [x = \ell_1]\}, \ell_3 = 2, \ell_4 = 0]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ 1\ \square\ \square\ \square\ \square, \\
 & \quad [x = \ell_1, y = \ell_4] :: [f = \ell_2, x = \ell_3] :: [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = 1, \ell_2 = \{\lambda y.x, [x = \ell_1]\}, \ell_3 = 2, \ell_4 = 0]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ 1\ \square\ \square\ \square, \quad [f = \ell_2, x = \ell_3] :: [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = 1, \ell_2 = \{\lambda y.x, [x = \ell_1]\}, \ell_3 = 2, \ell_4 = 0]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ 1\ \square\ \square, \quad [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = 1, \ell_2 = \{\lambda y.x, [x = \ell_1]\}, \ell_3 = 2, \ell_4 = 0]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ 1\ \square, \quad [x = \ell_1] :: [\,], [\ell_1 = 1, \ell_2 = \{\lambda y.x, [x = \ell_1]\}, \ell_3 = 2, \ell_4 = 0]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ 1, \quad [\,], [\ell_1 = 1, \ell_2 = \{\lambda y.x, [x = \ell_1]\}, \ell_3 = 2, \ell_4 = 0]\ \rangle
\end{array}
$$

Example 5. $(\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0) \to_{\mathsf{cl}}^{*} 2$

Proof.

$$
\begin{array}{lll}
 & \langle\ \mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0, & [\,],[\,]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ \{\lambda x.\mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0, [\,]\}\ 1, & [\,],[\,]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ \mathsf{let}\ f = \lambda y.x\ \mathsf{in}\ x := 2; f\ 0\ \Box, & [x = \ell_1] :: [\,],[\ell_1 = 1]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ \{\lambda f.x := 2; f\ 0, [x = \ell_1]\}\ \lambda y.x\ \Box, & [x = \ell_1] :: [\,],[\ell_1 = 1]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ \{\lambda f.x := 2; f\ 0, [x = \ell_1]\}\ \{\lambda y.x, [x = \ell_1]\}\ \Box, & [x = \ell_1] :: [\,],[\ell_1 = 1]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ x := 2; f\ 0\ \Box\ \Box, & [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], \\
 & & [\ell_1 = 1, \ell_2 = \{\lambda y.x, [x = \ell_1]\}]\ \rangle \\
\to^*_{\mathsf{cl}} & \langle\ f\ 0\ \Box\ \Box, [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], [\ell_1 = 2, \ell_2 = \{\lambda y.x, [x = \ell_1]\}]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ \{\lambda y.x, [x = \ell_1]\}\ 0\ \Box\ \Box, & [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], \\
 & & [\ell_1 = 2, \ell_2 = \{\lambda y.x, [x = \ell_1]\}]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ x\ \Box\ \Box\ \Box, & [x = \ell_1, y = \ell_3] :: [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], \\
 & & [\ell_1 = 2, \ell_2 = \{\lambda y.x, [x = \ell_1]\}, \ell_3 = 0]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ 2\ \Box\ \Box\ \Box, & [x = \ell_1, y = \ell_3] :: [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], \\
 & & [\ell_1 = 2, \ell_2 = \{\lambda y.x, [x = \ell_1]\}, \ell_3 = 0]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ 2\ \Box\ \Box, & [x = \ell_1, f = \ell_2] :: [x = \ell_1] :: [\,], \\
 & & [\ell_1 = 2, \ell_2 = \{\lambda y.x, [x = \ell_1]\}, \ell_3 = 0]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ 2\ \Box, & [x = \ell_1] :: [\,], [\ell_1 = 2, \ell_2 = \{\lambda y.x, [x = \ell_1]\}, \ell_3 = 0]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ 2, & [\,], [\ell_1 = 2, \ell_2 = \{\lambda y.x, [x = \ell_1]\}, \ell_3 = 0]\ \rangle
\end{array}
$$

Example 6. $(\mathsf{let\,rec}\ f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n\ \mathsf{in}\ f\ 3) \to^*_{\mathsf{cl}} 6$

Proof. This example is particularly interesting as it shows how nested $\Box$ allow to interpret the same variable in different scopes. In all the example e stands for $\{\lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n, [f = \ell_1]\}$, and d stands for $\{\lambda n.n, [\,]\}$, a

dummy value used when creating the recursive function f.

$$
\begin{array}{ll}
 & \langle\ \mathsf{let\,rec}\ f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n\ \mathsf{in}\ f\ 3, \quad [\,], [\,]\ \rangle \\
\to^*_{\mathsf{cl}} & \langle\ f := \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n; f\ 3\ \Box, \quad [f = \ell_1] :: [\,], [\ell_1 = d]\ \rangle \\
\to^*_{\mathsf{cl}} & \langle\ f\ 3\ \Box, \quad [f = \ell_1] :: [\,], [\ell_1 = e]\ \rangle \\
\to^*_{\mathsf{cl}} & \langle\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(n-1) \times n\ \Box\ \Box, \quad [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = e, \ell_2 = 3]\ \rangle \\
\to^*_{\mathsf{cl}} & \langle\ (f\ 2) \times n\ \Box\ \Box, \quad [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = e, \ell_2 = 3]\ \rangle \\
\to^*_{\mathsf{cl}} & \langle\ (\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times f(n-1)\ \Box) \times n\ \Box\ \Box, \\
 & \qquad [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = e, \ell_2 = 3, \ell_3 = 2]\ \rangle \\
\to^*_{\mathsf{cl}} & \langle\ ((f\ 1) \times n\ \Box) \times n\ \Box\ \Box, \quad [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = e, \ell_2 = 3, \ell_3 = 2]\ \rangle \\
\to^*_{\mathsf{cl}} & \langle\ ((\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times f(n-1)\ \Box) \times n\ \Box) \times n\ \Box\ \Box, \\
 & \qquad [f = \ell_1, n = \ell_4] :: [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1]\ \rangle \\
\to^*_{\mathsf{cl}} & \langle\ (((f\ 0) \times n\ \Box) \times n\ \Box) \times n\ \Box\ \Box, \\
 & \qquad [f = \ell_1, n = \ell_4] :: [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1]\ \rangle \\
\to^*_{\mathsf{cl}} & \langle\ (((\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times f(n-1)\ \Box) \times n\ \Box) \times n\ \Box) \times n\ \Box\ \Box, \\
 & \qquad [f = \ell_1, n = \ell_5] :: [f = \ell_1, n = \ell_4] :: [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: \\
 & \qquad [f = \ell_1] :: [\,], [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ (((\mathsf{if}\ 0 = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times f(n-1)\ \Box) \times n\ \Box) \times n\ \Box) \times n\ \Box\ \Box, \\
 & \qquad [f = \ell_1, n = \ell_5] :: [f = \ell_1, n = \ell_4] :: [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: \\
 & \qquad [f = \ell_1] :: [\,], [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ (((1\ \Box) \times n\ \Box) \times n\ \Box) \times n\ \Box\ \Box, \\
 & \qquad [f = \ell_1, n = \ell_5] :: [f = \ell_1, n = \ell_4] :: [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: \\
 & \qquad [f = \ell_1] :: [\,], [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle \\
\to_{\mathsf{cl}} & \langle\ ((1 \times n\ \Box) \times n\ \Box) \times n\ \Box\ \Box, \\
 & \qquad [f = \ell_1, n = \ell_4] :: [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle \\
\to^*_{\mathsf{cl}} & \langle\ ((1 \times 1) \times n\ \Box) \times n\ \Box\ \Box, [f = \ell_1, n = \ell_3] :: [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle \\
\to^*_{\mathsf{cl}} & \langle\ ((1 \times 1) \times 2) \times n\ \Box\ \Box, \quad [f = \ell_1, n = \ell_2] :: [f = \ell_1] :: [\,], \\
 & \qquad [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle \\
\to^*_{\mathsf{cl}} & \langle\ 6, \quad [\,], [\ell_1 = e, \ell_2 = 3, \ell_3 = 2, \ell_4 = 1, \ell_5 = 0]\ \rangle
\end{array}
$$

4 Equivalence of the small-step semantics

4.1 Definitions

There is a very strong correspondence between the semantics of closures and capsules. To give a precise account of this correspondence, we introduce an injective partial function $h : \mathsf{Loc} \rightharpoonup \mathsf{Var}$ with which we define three relations. Each relation is between an element of the semantics of closures and an element of the semantics of capsules that play similar roles:

- $v \xrightarrow{h} i$ between values and irreducible terms;
- $\mu \xrightarrow{h} \gamma$ between stores and capsule environments;
- $(s,\, \Sigma,\, \mu) \overset{h}{\sim} \langle e,\, \gamma \rangle$ between states and capsules;

One thing to notice is that nothing in the semantics of capsules plays the same role as the stack of environments Σ in the semantics of closures: capsule environments γ relate to memories μ, and the stack of environments Σ has been simplified. Let us now give precise definitions of those relations.

Definition 1. *Given a value v and an irreducible term i, we say that* h transforms v into i*, where h is an injective map $h : \mathsf{Loc} \rightharpoonup \mathsf{Var}$, and we write $v \xrightarrow{h} i$, if and only if:*

- $v = i$ *when* $v \in \mathsf{Const}$*, or*
- $\mathsf{codom}\, \tau \subseteq \mathsf{dom}\, h$ *and* $(h \circ \tau)(\lambda x.a) = i$ *when* $v = \{\lambda x.a,\, \tau\} \in \mathsf{Cl}$

Definition 2. *Given a store μ and a capsule environment γ, we say that h* transforms μ into γ*, where h is an injective map $h : \mathsf{Loc} \rightharpoonup \mathsf{Var}$, and we write $\mu \xrightarrow{h} \gamma$, if and only if:*

$$\mathsf{dom}\, h = \mathsf{dom}\, \mu \qquad h(\mathsf{dom}\, \mu) = \mathsf{dom}\, \gamma$$
$$\forall \ell \in \mathsf{dom}\, \mu,\, \mu(\ell) \xrightarrow{h} \gamma(h(\ell))$$

We are now ready to give a precise account of the interpretation of variables in state environments, as described in Sect. 3.2. Given a map $h : \mathsf{Loc} \rightharpoonup \mathsf{Var}$ and a stack of environments Σ, let us inductively define the operator $h \circ \Sigma : \mathsf{StExp} \rightarrow$

Exp as:

$$
\begin{aligned}
h \circ (\Sigma :: \sigma)(e) &= h \circ \sigma(e) \\
h \circ \Sigma(\{\lambda x.a,\, \sigma\}) &= \sigma(\lambda x.a) \\
h \circ (\Sigma :: \sigma)(s\ \square) &= h \circ \Sigma(s) \\
h \circ \Sigma(s\ e) &= (h \circ \Sigma(s))\ (h \circ \Sigma(e)) \\
h \circ \Sigma(v\ s) &= (h \circ \Sigma(v))\ (h \circ \Sigma(s)) \\
h \circ (\Sigma :: \sigma)(x := s) &= (h \circ \sigma(x)) := h \circ (\Sigma :: \sigma)(s) \\
h \circ \Sigma(s; e) &= h \circ \Sigma(s); h \circ \Sigma(e) \\
h \circ \Sigma(\mathsf{if}\ s\ \mathsf{then}\ d\ \mathsf{else}\ e) &= \mathsf{if}\ h \circ \Sigma(s)\ \mathsf{then}\ h \circ \Sigma(d)\ \mathsf{else}\ h \circ \Sigma(e)
\end{aligned}
$$

Definition 3. *Given a state* $(s,\, \Sigma,\, \mu)$ *and a capsule* $\langle e,\, \gamma \rangle$*, both valid, we say that they are* bisimilar under h*, where* h *is an injective map* $h : \mathsf{Loc} \rightharpoonup \mathsf{Var}$*, and we write* $(s,\, \Sigma,\, \mu) \overset{h}{\sim} \langle e,\, \gamma \rangle$*, if and only if*

$$h \circ \Sigma(s) = e \qquad\qquad \mu \overset{h}{\to} \gamma$$

4.2 Soundness of Capsules with respect to Closures

Now that we know how to relate each element of both semantics, Theorem 1 shows that any derivation using closures mirrors zero or more derivation steps using capsules, and Theorem 2 shows that any derivation step using capsules mirrors zero or more derivation steps using closures. Combined, they give rise to Corollary 1, which shows that any derivation using capsules is mirrored by a derivation using closures, and vice-versa.

Theorem 1. *If* $(s,\, \Sigma,\, \mu) \overset{h}{\sim} \langle d,\, \gamma \rangle$ *and* $(s,\, \Sigma,\, \mu) \to_{\mathsf{cl}} (t,\, \Pi,\, \nu)$*, then there exists* e, δ *such that*

$$\langle d,\, \gamma \rangle \to^*_{\mathsf{ca}} \langle e,\, \delta \rangle \qquad\qquad (t,\, \Pi,\, \nu) \overset{g}{\sim} \langle e,\, \delta \rangle$$

where g *is an extension of* h*, i.e.,* $\mathsf{dom}\, h \subseteq \mathsf{dom}\, g$ *and* h *and* g *agree on* $\mathsf{dom}\, h$*.*

Theorem 2. *If* $(s,\, \Sigma,\, \mu) \overset{h}{\sim} \langle d,\, \gamma \rangle$ *and* $\langle d,\, \gamma \rangle \to_{\mathsf{ca}} \langle e,\, \delta \rangle$*, then there exists* t, Π, ν *such that*

$$(s,\, \Sigma,\, \mu) \to^*_{\mathsf{cl}} (t,\, \Pi,\, \nu) \qquad\qquad (t,\, \Pi,\, \nu) \overset{g}{\sim} \langle e,\, \delta \rangle$$

where g *is an extension of* h*, i.e.,* $\mathsf{dom}\, h \subseteq \mathsf{dom}\, g$ *and* h *and* g *agree on* $\mathsf{dom}\, h$*.*

Corollary 1. *If* $(s,\, \Sigma,\, \mu) \overset{h}{\sim} \langle d,\, \gamma \rangle$ *then*

– *if* $(s,\, \Sigma,\, \mu) \to_{\mathsf{cl}}^{*} (t,\, \Pi,\, \nu)$ *then there exists* e, δ *such that*

$$\langle d,\, \gamma\rangle \to_{\mathsf{ca}}^{*} \langle e,\, \delta\rangle \qquad\qquad (t,\, \Pi,\, \nu) \overset{g}{\sim} \langle e,\, \delta\rangle$$

– *if* $\langle d,\, \gamma\rangle \to_{\mathsf{ca}}^{*} \langle e,\, \delta\rangle$ *then there exists* t, Π, ν *such that*

$$(s,\, \Sigma,\, \mu) \to_{\mathsf{cl}}^{*} (t,\, \Pi,\, \nu) \qquad\qquad (t,\, \Pi,\, \nu) \overset{g}{\sim} \langle e,\, \delta\rangle$$

where g *is an extension of* h*, i.e.,* $\mathsf{dom}\, h \subseteq \mathsf{dom}\, g$ *and* h *and* g *agree on* $\mathsf{dom}\, h$*.*

Proof of Corollary 1. We use standard arguments on weak bisimilarity. The first part is proved by recurrence on the number of steps of the derivation of $(s,\, \Sigma,\, \mu) \to_{\mathsf{cl}}^{*} (t,\, \Pi,\, \nu)$ and application of Theorem 1. Similarly, the second part is by recurrence on the number of steps of $\langle d,\, \gamma\rangle \to_{\mathsf{ca}}^{*} \langle e,\, \delta\rangle$ and application of Theorem 2. □

Proof of Theorem 1. We proceed by induction on the derivation of $(s,\, \Sigma,\, \mu) \to_{\mathsf{cl}} (t,\, \Pi,\, \nu)$. In the interest of space, we only show the most interesting cases of the induction in the main text: variable call x, λ-abstraction $\lambda x.e$, function application of a closure $\{\lambda x.a,\, \sigma\}\ v$, popping from the environment stack $v\ \square$, variable assignment $x := e$, contexts $C[s]$ and computing with the pop indicator $s\ \square$. The other cases, function application of a constant function $f\ c$, composition $d; e$, if conditional $\mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e$ and while loop $\mathsf{while}\ b\ \mathsf{do}\ e$, are straightforward inductive arguments.

Variable call If $s = x$ for some variable x and $\Sigma = [\sigma]$ then $d = (h \circ \sigma)(s) = y$ with y the variable such that $y = (h \circ \sigma)(x)$.

By definition of $\to_{\mathsf{cl}}$, $(t,\, \Pi,\, \nu) = (\mu(\sigma(x)),\, [\sigma],\, \mu)$, and by definition of $\to_{\mathsf{ca}}$, $\langle d,\, \gamma\rangle = \langle y,\, \gamma\rangle \to_{\mathsf{ca}} \langle \gamma(y),\, \gamma\rangle$. Moreover $\mu \overset{h}{\to} \gamma$, therefore by definition of $\overset{h}{\to}$, $\mu(\sigma(x)) \overset{h}{\to} \gamma(h(\sigma(x))) = \gamma(y)$. Therefore, with $g = h$, $(t,\, \Pi,\, \nu) = (\mu(\sigma(x)),\, [\sigma],\, \mu) \overset{g}{\sim} \langle \gamma(y),\, \gamma\rangle$ which completes this case.

λ-Abstraction If $s = \lambda x.a$ and $\Sigma = [\sigma]$, then $d = (h \circ \sigma)(\lambda x.a)$ which is a term α-equivalent to s, so $d = \lambda x.b$ for some b. Indeed, the variable x does not change from s to d since only the free variables of s are affected by $h \circ \sigma$.

By definition of $\to_{\mathsf{cl}}$, $(t,\, \Pi,\, \nu) = (\{\lambda x.a,\, \sigma\},\, [\sigma],\, \mu)$, and by reflexivity of $\to_{\mathsf{ca}}^{*}$, $\langle d,\, \gamma\rangle = \langle \lambda x.b,\, \gamma\rangle \to_{\mathsf{ca}}^{*} \langle \lambda x.b,\, \gamma\rangle$. But $\mathsf{codom}\, \sigma \subseteq \mathsf{dom}\, h$ and $\lambda x.b = (h \circ \sigma)(\lambda x.a)$, therefore $\{\lambda x.a,\, \sigma\} \overset{h}{\to} \lambda x.b$. Moreover we know $\mu \overset{h}{\to} \gamma$ and with $g = h$, we get $(\{\lambda x.a,\, \sigma\},\, [\sigma],\, \mu) \overset{g}{\sim} \langle \lambda x.b,\, \gamma\rangle$ which completes this case.

Function application of a closure If $s = \{\lambda x.a,\, \sigma\}\ v$ and $\Sigma = [\tau]$, then $(h \circ \Sigma)(\{\lambda x.a,\, \sigma\}) = (h \circ \sigma)(\lambda x.a)$ is a $\lambda x.b$ for some expression b, and $(h \circ \Sigma)(v)$ is some irreducible term i. Since $d = (h \circ \Sigma)(s)$, $d = (\lambda x.b)\ i$.

By definition of $\to_{\mathsf{cl}}$, $(t,\, \Pi,\, \nu) = (a\ \square,\, \sigma[x/\ell] :: \tau,\, \mu[\ell/v])$ with ℓ fresh, and by definition of $\to_{\mathsf{ca}}$, $\langle d,\, \gamma\rangle \to_{\mathsf{ca}} \langle b[x/y],\, \gamma[y/i]\rangle$, with y fresh. Let $g : \mathsf{Loc} \rightharpoonup \mathsf{Var}$ such that:

$$\begin{gathered} g : \mathsf{dom}\, h \cup \{\ell\} \to \mathsf{codom}\, g \cup \{y\} \\ \ell_h \in \mathsf{dom}\, h \mapsto h(\ell_h) \\ \ell \mapsto y \end{gathered}$$

Lemma 1. $(a,\, [\sigma[x/\ell]],\, \mu[\ell/v]) \overset{g}{\sim} \langle b[x/y],\, \gamma[y/i]\rangle$

Proof. First of all, $\lambda x.b = (h \circ \sigma)(\lambda x.a)$, g is an extension of h and $\mathsf{FV}(\lambda x.a) \subseteq \mathsf{dom}\, h$, therefore $\lambda x.b = (g \circ \sigma)(\lambda x.a)$. Now $b[x/y] = ((g \circ \sigma)[x/y])(a) = (g \circ \sigma[x/\ell])(\lambda x.a)$ since $g(\ell) = y$.

We further need to argue that $\mu[\ell/v] \overset{g}{\to} \gamma[y/i]$. We already know that $\mathsf{dom}\, g = \mathsf{dom}\, h \cup \{\ell\} = \mathsf{dom}\, \mu \cup \{\ell\} = \mathsf{dom}\, \mu[\ell/v]$, and $g(\mathsf{dom}\, \mu[\ell/v]) = \mathsf{codom}\, h \cup \{y\} = \mathsf{dom}\, \gamma[y/i]$. Let $\ell' \in \mathsf{dom}\, \mu[\ell/v]$. If $\ell' \in \mathsf{dom}\, \mu$, then $\mu[\ell/v](\ell') = \mu(\ell') \overset{h}{\to} \gamma(g(\ell')) = \gamma[y/i](g(\ell'))$ by injectivity of g, therefore $\mu[\ell/v](\ell') \overset{g}{\to} \gamma[y/i](g(\ell'))$. Otherwise, $\ell' = \ell$ and then $\mu[\ell/v](\ell) = v \overset{h}{\to} i = \gamma[y/i](y) = \gamma[y/i](g(\ell))$, therefore since g is an extension of h, $\mu[\ell/v](\ell) \overset{g}{\to} \gamma[y/i](g(\ell))$. This completes the proof of the lemma.

Using lemma 1, we get that $(g \circ [\sigma[x/\ell]])(a) = b[x/y]$ and $\mu[\ell/v] \overset{g}{\to} \gamma[y/i]$. But $g \circ (\sigma[x/\ell] :: \tau)(a\ \square) = (g \circ [\sigma[x/\ell]])(a)$, therefore $(a\ \square,\, \sigma[x/\ell] :: \tau,\, \mu[\ell/v]) \overset{g}{\to} \gamma[y/i]$, which completes this case.

Popping from the environment stack If $s = v\ \square$ for some value v and $\Sigma = \sigma :: \tau$, then $d = (h \circ \Sigma)(v\ \square) = (h \circ \Sigma)(v)$, which is an irreducible term i such that $v \overset{h}{\to} i$, since:

- if v is a constant c, $i = (h \circ \Sigma)(c) = c$;
- if v is a closure $\{\lambda x.a,\, \sigma'\}$, $i = (h \circ \Sigma)(\{\lambda x.a,\, \sigma'\}) = (h \circ \sigma')(\lambda x.a)$ and $\mathsf{codom}\, \sigma' \subseteq \mathsf{dom}\, h$.

By definition of $\to_{\mathsf{cl}}$, $(t,\, \Pi,\, \nu) = (v,\, [\tau],\, \mu)$, and by reflexivity of $\to^*_{\mathsf{ca}}$, $\langle d,\, \gamma\rangle = \langle i,\, \gamma\rangle \to^*_{\mathsf{ca}} \langle i,\, \gamma\rangle$. But $i = (h \circ \Sigma(v))$ does not depend on Σ, therefore $i = (h \circ [\tau])(v)$. Moreover we know $\mu \overset{h}{\to} \gamma$ and with $g = h$, we get $(v,\, [\tau],\, \mu) \overset{g}{\sim} \langle i,\, \gamma\rangle$ which completes this case.

Variable assignment If $s = (x := v)$ for some variable x and value v and $\Sigma = [\sigma]$, then $d = (h \circ \Sigma)(x := v) = (y := i)$ with y a variable such that $y = (h \circ \sigma)(x)$ and $i = (h \circ \Sigma)(v)$. Therefore $(v,\, \sigma,\, \mu) \overset{h}{\sim} \langle i,\, \gamma\rangle$.

By definition of $\to_{\mathsf{cl}}$, $(s,\, \Pi,\, \nu) = ((),\, [\sigma],\, \mu[\sigma(x)/v])$, and by definition of $\to_{\mathsf{ca}}$, $\langle d,\, \gamma\rangle = \langle y := i,\, \gamma\rangle = \langle (),\, \gamma[y/i]\rangle$. The following lemma completes this case.

Lemma 2. $((), \sigma, \mu[\sigma(x)/v]) \overset{h}{\sim} \langle (), \gamma[y/i] \rangle$

Proof. The domain conditions are fulfilled since $(v, \sigma, \mu) \overset{h}{\sim} \langle i, \gamma \rangle$, $\mathsf{dom}\,\mu = \mathsf{dom}\,\mu[\sigma(x)/v]$ and $\mathsf{dom}\,\gamma = \mathsf{dom}\,\gamma[y/i]$. Let $\ell \in \mathsf{dom}\,\mu[\sigma(x)/v] = \mathsf{dom}\,\mu$. If $\ell = \sigma(x)$ then $\mu[\sigma(x)/v](\ell) = v \overset{h}{\sim} i = \gamma[y/i](y) = \gamma[y/i](h(\ell))$ since $h(\ell) = (h \circ \sigma)(x) = (h \circ \sigma)(x) = y$. Otherwise $\mu[\sigma(x)/v](\ell) = \mu(\ell) \overset{h}{\sim} \gamma(h(\ell)) = \gamma[y/i](h(\ell))$ using that h is injective and h is an extension of h. Finally $() \overset{h}{\to} ()$, which completes the proof of the lemma.

Contexts If $s = C[s_1]$ for some context C such that $(s_1, \Sigma, \mu) \to_{\mathsf{cl}} (t_1, \Pi, \nu)$, then by definition of $\to_{\mathsf{cl}}$, $t = C[t_1]$. By definition of $\overset{h}{\sim}$ there exists d_1 such that $d = C[d_1]$. By induction hypothesis there exists e_1, δ such that $\langle d_1, \gamma \rangle \to^*_{\mathsf{ca}} \langle e_1, \delta \rangle$ and $(t_1, \Pi, \nu) \overset{g}{\sim} \langle e_1, \delta \rangle$ for some g extension of h. By definition of $\to_{\mathsf{ca}}$, $\langle C[d_1], \gamma \rangle \to^*_{\mathsf{ca}} \langle C[e_1], \delta \rangle$. By induction on the structure of C, and using the fact that the context C cannot contain any $\Box$, we can then prove that $(C[t_1], \Pi, \nu) \overset{g}{\sim} \langle C[e_1], \delta \rangle$, which completes this case.

Computing under the pop indicator $\Box$ If $s = s_1\ \Box$ for s_1 not a value, such that $(s_1, \Sigma, \mu) \to_{\mathsf{cl}} (t_1, \Pi, \nu)$, and $\Sigma = \Sigma' :: \sigma$, then by definition of $\to_{\mathsf{cl}}$, $t = t_1\ \Box$ and $\Pi = \Pi' :: \sigma$ for some Π'. $(s_1\ \Box, \Sigma' :: \sigma, \mu) \overset{h}{\sim} \langle d, \gamma \rangle$, therefore $(s_1, \Sigma', \mu) \overset{h}{\sim} \langle d, \gamma \rangle$. By induction hypothesis there exists e, δ such that $\langle d, \gamma \rangle \to^*_{\mathsf{ca}} \langle e, \delta \rangle$ and $(t_1, \Pi, \nu) \overset{h}{\sim} \langle e, \delta \rangle$. Now this proves that $(t_1\ \Box, \Pi' :: \sigma, \nu) \overset{h}{\sim} \langle e, \delta \rangle$, which completes this case.

$\Box$

Proof of Theorem 2. We proceed similarly as for the proof of Theorem 1, by induction on the derivation of $\langle d, \gamma \rangle \to_{\mathsf{ca}} \langle e, \delta \rangle$. We do not detail any case here. The cases for variable call, function application of a λ-term, variable assignment, and contexts are symmetric to the ones seen in the proof of Theorem 1. The case for function application of a constant function, composition, if conditional and while loop are straightforward inductive arguments. Finally, this Theorem does not need cases for λ-abstractions, popping from the environment stack or computing with the pop indicator, as no rule in $\to_{\mathsf{ca}}$ applies to those. $\Box$

5 Discussion

5.1 Capsules and Closures: a strong correspondence

Corollary 1 shows that capsules and closures are very strongly related. Not only there is a derivation based on capsules for every derivation based on closures, but these two derivations mirror each other. The computations are completely bisimilar, even though definining computations for capsules is simpler.

5.2 Capsules allow to suppress the stack of environments Σ

When using closures, a state is a triple $(s,\ \Sigma,\ \mu)$ whereas when using capsules, it is just a capsule $\langle e,\ \gamma\rangle$. If they are bisimilar under h, it means that $(h \circ \Sigma)(d) = e$ and $\mu \xrightarrow{h} \gamma$. Capsules eliminate the need for the stack of environments Σ and thus suppress the indirection in closures that was needed to handle imperative features. Their small-step semantics also does not need any stack of environments of any sort, making the state of computation much simpler. Finally, we originally created the capsule environment γ to replace the (closure) environments of Σ. However, it is remarkable that γ is much closer to the store μ, while at the same time eliminating the need for Σ.

Acknowledgements

We would like to thank Nikhil Swamy for suggesting this work during a presentation of [8].

References

1. Aboul-Hosn, K.: Programming with private state. Honors Thesis, The Pennsylvania State University (December 2001), `http://www.cs.cornell.edu/%7Ekamal/thesis.pdf`
2. Aboul-Hosn, K., Kozen, D.: Relational semantics of local variable scoping. Tech. Rep. 2005-2000, Cornell University (2005), `http://www.cs.cornell.edu/%7Ekamal/local.pdf`
3. Aboul-Hosn, K., Kozen, D.: Relational semantics for higher-order programs. In: Uustalu, T. (ed.) Proc. 8th Int. Conf. Mathematics of Program Construction (MPC'06). Lecture Notes in Computer Science, vol. 4014, pp. 29–48. Springer (July 2006)
4. Abramsky, S., Honda, K., McCusker, G.: A fully abstract game semantics for general references. In: LICS '98: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science. pp. 334–344. IEEE Computer Society, Washington, DC, USA (1998)
5. Abramsky, S., McCusker, G.: Linearity, sharing and state: a fully abstract game semantics for idealized ALGOL with active expressions. Electr. Notes Theor. Comput. Sci. 3 (1996)
6. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theoretical Computer Science 103, 235–271 (1992)
7. Halpern, J.Y., Meyer, A.R., Trakhtenbrot, B.A.: The semantics of local storage, or what makes the free-list free? In: Proc. 11th ACM Symp. Principles of Programming Languages (POPL'84). pp. 245–257. New York, NY, USA (1984)
8. Jeannin, J.B.: Capsules and closures. In: Mislove, M., Ouaknine, J. (eds.) Proc. 27th Conf. Math. Found. Programming Semantics (MFPS XXVII). Elsevier Electronic Notes in Theoretical Computer Science, Pittsburgh, PA (May 2011)

9. Jeannin, J.B., Kozen, D.: Computing with capsules. Tech. Rep. http://hdl.handle.net/1813/22082, Computing and Information Science, Cornell University (January 2011)
10. Laird, J.: A game semantics of local names and good variables. In: Walukiewicz, I. (ed.) FoSSaCS. Lecture Notes in Computer Science, vol. 2987, pp. 289–303. Springer (2004)
11. Mason, I.A., Talcott, C.L.: References, local variables and operational reasoning. In: Seventh Annual Symposium on Logic in Computer Science. pp. 186–197. IEEE (1992), http://www-formal.stanford.edu/MT/92lics.ps.Z
12. Mason, I., Talcott, C.: Programming, transforming, and proving with function abstractions and memories
13. Mason, I., Talcott, C.: Axiomatizing operational equivalence in the presence of side effects. In: Fourth Annual Symposium on Logic in Computer Science. IEEE. pp. 284–293. IEEE Computer Society Press (1989)
14. Mason, I., Talcott, C.: Equivalence in functional languages with effects (1991)
15. Milne, R., Strachey, C.: A Theory of Programming Language Semantics. Halsted Press, New York, NY, USA (1977)
16. Moggi, E.: Notions of computation and monads. Information and Computation 93(1) (1991)
17. Pitts, A.M.: Operationally-based theories of program equivalence. In: Dybjer, P., Pitts, A.M. (eds.) Semantics and Logics of Computation, pp. 241–298. Publications of the Newton Institute, Cambridge University Press (1997), http://www.cs.tau.ac.il/~nachumd/formal/exam/pitts.pdf
18. Pitts, A.M., Stark, I.D.B.: Operational reasoning in functions with local state. In: Gordon, A.D., Pitts, A.M. (eds.) Higher Order Operational Techniques in Semantics, pp. 227–273. Cambridge University Press (1998), http://homepages.inf.ed.ac.uk/stark/operfl.pdf
19. Pitts, A.M., Stark, I.D.B.: Observable properties of higher order functions that dynamically create local names, or what's new? In: Borzyszkowski, A.M., Sokolowski, S. (eds.) MFCS. Lecture Notes in Computer Science, vol. 711, pp. 122–141. Springer (1993)
20. Scott, D.: Mathematical concepts in programmng language semantics. In: Proc. 1972 Spring Joint Computer Conferences. pp. 225–34. AFIPS Press, Montvale, NJ (1972)
21. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, MA, USA (1981)
22. Sussman, G.J., Steele, G.L.: Scheme: A interpreter for extended lambda calculus. Higher-Order and Symbolic Computation 11, 405–439 (1998), http://dx.doi.org/10.1023/A:1010035624696, 10.1023/A:1010035624696
23. Winskel, G.: The Formal Semantics of Programming Languages. MIT Press (1993)