Defunctionalized Interpreters for Call-by-Need Evaluation

Olivier Danvy[1], Kevin Millikin[2], Johan Munk[3], and Ian Zerny[1]

[1] Department of Computer Science, Aarhus University
Aabogade 34, 8200 Aarhus N, Denmark — {danvy,zerny}@cs.au.dk
[2] Google
Aabogade 15, 8200 Aarhus N, Denmark — kmillikin@google.com
[3] Arctic Lake Systems
Aabogade 15, 8200 Aarhus N, Denmark — johanmunk@gmail.com

Abstract. Starting from the standard call-by-need reduction for the λ-calculus that is common to Ariola, Felleisen, Maraist, Odersky, and Wadler, we inter-derive a series of hygienic semantic artifacts: a reduction-free stateless abstract machine, a continuation-passing evaluation function, and what appears to be the first heapless natural semantics for call-by-need evaluation. Furthermore we observe that a data structure and a judgment in this natural semantics are in defunctionalized form. The refunctionalized counterpart of this evaluation function is an extended direct semantics in the sense of Cartwright and Felleisen.
Overall, the semantic artifacts presented here are simpler than many other such artifacts that have been independently worked out, and which require ingenuity, skill, and independent soundness proofs on a case-by-case basis. They are also simpler to inter-derive because the inter-derivational tools (e.g., refocusing and defunctionalization) already exist.

1 Introduction

A famous functional programmer once was asked to give an overview talk. He began with “This talk is about lazy functional programming and call by need.” and paused. Then, quizzically looking at the audience, he quipped: “Are there any questions?” There were some, and so he continued: “Now listen very carefully, I shall say this only once.”

This apocryphal story illustrates demand-driven computation and memoization of intermediate results, two key features that have elicited a fascinating variety of semantic specifications and implementation techniques over the years, ranging from purely syntactic treatments to mutable state, and featuring small-step operational semantics [2,26], a range of abstract machines [17,19,33], big-step operational semantics [1,24], as well as evaluation functions [20,22].

In this article, we extract the computational content of the standard call-by-need reduction for the λ-calculus that is common to both Ariola and Felleisen [2] and Maraist, Odersky, and Wadler [26]. This computational content takes the forms of a one-step reduction function, an abstract machine, and a natural semantics that are mutually compatible and all abide by Barendregt’s variable convention [4, page 26]. Rather than handcrafting each of these semantic artifacts from scratch and then proving a series of soundness theorems, we successively inter-derive them from small steps to big steps using a series of fully correct transformations. To this end, we follow the programme outlined in the first author's invited talk at ICFP 2008 [9]. To this programme we add one new refunctionalization step that is specific to call by need:

0. we modify one axiom in the standard call-by-need reduction to make the associated one-step reduction function preserve Barendregt's variable convention;
1. iterating this hygienic one-step reduction function yields a reduction-based evaluation function, which we refocus to obtain a reduction-free evaluation function with the same built-in hygiene; this evaluation function takes the form of an abstract machine and is correct by construction;
2. we simplify this hygienic abstract machine by compressing its corridor transitions, and we refunctionalize this simplified hygienic abstract machine into a continuation-passing evaluation function, which we write back to direct style, obtaining a functional program that is correct by construction and that implements a heapless natural semantics with the same built-in hygiene;
3. in addition, we observe that a data structure and a judgment in this hygienic natural semantics are in defunctionalized form, and we present the corresponding higher-order evaluation function.

The ML code of the entire derivation is available from the authors.

Prerequisites: We assume the reader to know the formats of a reduction semantics, an abstract machine, and a natural semantics as can be gathered, e.g., in the first author's lecture notes at AFP 2008 [10].

2 The standard call-by-name reduction for the $\boldsymbol{\lambda}$-calculus

The standard call-by-name reduction for the λ_{let}-calculus is a simplification of Ariola et al.'s call-by-need formulation presented in Section 3. This call-by-name formulation reads as follows:

Definition 1 (call-by-name λ_{let}-calculus).

Syntax:

$$\begin{array}{rcl} \textsf{\textit{Terms}} \ni T & ::= & x \mid \lambda x.T \mid T\ T \mid \textsf{let}\ x\ \textsf{be}\ T\ \textsf{in}\ T \\ \textsf{\textit{Values}} \ni V & ::= & \lambda x.T \\ \textsf{\textit{Answers}} \ni A & ::= & V \mid \textsf{let}\ x\ \textsf{be}\ T\ \textsf{in}\ A \\ \textsf{\textit{Evaluation Contexts}} \ni E & ::= & [\,] \mid E\ T \mid \textsf{let}\ x\ \textsf{be}\ T\ \textsf{in}\ E \end{array}$$

Axioms (i.e., contraction rules):[1]

$$\begin{array}{lrcl} (I) & (\lambda x.T)\ T_1 & \rightarrow & \textsf{let}\ x\ \textsf{be}\ T_1\ \textsf{in}\ T \\ (N) & \textsf{let}\ x\ \textsf{be}\ T\ \textsf{in}\ E[x] & \rightarrow & \textsf{let}\ x\ \textsf{be}\ T\ \textsf{in}\ E[T] \\ (C) & (\textsf{let}\ x\ \textsf{be}\ T_1\ \textsf{in}\ A)\ T_2 & \rightarrow & \textsf{let}\ x\ \textsf{be}\ T_1\ \textsf{in}\ A\ T_2 \end{array}$$

[1] The unusual notation "$E[x]$" in Rule (N) stands for a term that uniquely decomposes into an evaluation context E and a variable x. Naturally, there may be more than one occurrence of x in the term $E[x]$.

In words: Terms are pure λ-terms with let expressions. Values are λ-abstractions. Answers are let expressions nested around a value. Evaluation contexts are terms with a hole and are constructed according to the call-by-name reduction strategy. As for the axioms, Rule (I) introduces a let binding from an application; Rule (N) hygienically substitutes a term for the occurrence of a let-bound variable arising in an evaluation context; and Rule (C) allows let bindings to commute with applications, hygienically, i.e., renaming what needs to be renamed so that no free variable is captured.

The following reduction sequence illustrates the demand-driven aspect of call by name as well as the duplication of work it entails, noting one-step reduction with $\mapsto_{\text{name}}$ and annotating each reduction step with the corresponding axiom:

$$\begin{array}{lr}
(\lambda z.z\ z)\ ((\lambda y.y)\ (\lambda x.x)) \mapsto_{\text{name}} & (I) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \colorbox{lightgray}{z}\ z \mapsto_{\text{name}} & (N) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ ((\lambda y.y)\ (\lambda x.x))\ z \mapsto_{\text{name}} & (I) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ (\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ y)\ z \mapsto_{\text{name}} & (C) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \colorbox{lightgray}{y}\ z \mapsto_{\text{name}} & (N) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ (\lambda x.x)\ z \mapsto_{\text{name}} & (I) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ z\ \mathsf{in}\ \colorbox{lightgray}{x} \mapsto_{\text{name}} & (N) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ z\ \mathsf{in}\ \colorbox{lightgray}{z} \mapsto_{\text{name}} & (N) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ z\ \mathsf{in}\ (\lambda y.y)\ (\lambda x.x) \mapsto_{\text{name}} & (I) \\
\ldots &
\end{array}$$

We have shaded the occurrences of the variables whose value is needed in the course of the reduction. Each of the two shaded occurrences of z triggers the reduction of $(\lambda y.y)\ (\lambda x.x)$. The result of this demand-driven reduction is not memoized.

3 The standard call-by-need reduction for the λ-calculus

Our starting point is the standard call-by-need reduction for the λ_{let}-calculus that is common to Ariola, Felleisen, Maraist, Odersky, and Wadler's articles [2, 26], renaming non-terminals for notational uniformity, and assuming that initial terms are closed:

Definition 2 (call-by-need λ_{let}-calculus).

Syntax:

$$\begin{array}{rcl}
\textsf{Terms} \ni T & ::= & x \mid \lambda x.T \mid T\ T \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ T \\
\textsf{Values} \ni V & ::= & \lambda x.T \\
\textsf{Answers} \ni A & ::= & V \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A \\
\textsf{Evaluation Contexts} \ni E & ::= & [\,] \mid E\ T \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ E \mid \mathsf{let}\ x\ \mathsf{be}\ E\ \mathsf{in}\ E[x]
\end{array}$$

Axioms (i.e., contraction rules):

$$\begin{array}{lrcl}
(I) & (\lambda x.T)\ T_1 & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T \\
(V) & \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[x] & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[V] \\
(C) & (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A)\ T_2 & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A\ T_2 \\
(A) & \begin{array}{l}\mathsf{let}\ x\ \mathsf{be}\ \mathsf{let}\ y\ \mathsf{be}\ T_1 \\ \qquad\quad \mathsf{in}\ A \\ \mathsf{in}\ E[x]\end{array} & \rightarrow & \begin{array}{l}\mathsf{let}\ y\ \mathsf{be}\ T_1 \\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ A \\ \qquad \mathsf{in}\ E[x]\end{array}
\end{array}$$

In words: Terms are pure λ-terms with let expressions. Values are λ-abstractions. Answers are let expressions nested around a value. Evaluation contexts are terms with a hole and are constructed according to the call-by-need reduction strategy. As for the axioms, Rule (I) introduces a let binding from an application; Rule (V) substitutes a value for the occurrence of a let-bound variable arising in an evaluation context; Rule (C) allows let bindings to commute with applications; and Rule (A) re-associates let bindings.

Where call by name uses Axiom (N), call by need uses (V), ensuring that only values are duplicated. The reduction strategy thus also differs, so that the definiens of a needed variable is first reduced and this variable is henceforth declared to denote this reduct.

The following reduction sequence illustrates the demand-driven aspect of call by need as well as the memoization of intermediate results it enables, noting one-step reduction with $\mapsto_{\text{need}}$ (and specifying it precisely in Section 4.7):

$$\begin{array}{ll}
(\lambda z.z\ z)\ ((\lambda y.y)\ (\lambda x.x)) \mapsto_{\text{need}} & (I) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \boxed{z}\ z \mapsto_{\text{need}} & (I) \\
\mathsf{let}\ z\ \mathsf{be}\ (\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \boxed{y})\ \mathsf{in}\ z\ z \mapsto_{\text{need}} & (V) \\
\mathsf{let}\ z\ \mathsf{be}\ (\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \lambda x.x)\ \mathsf{in}\ z\ z \mapsto_{\text{need}} & (A) \\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \boxed{z}\ z \mapsto_{\text{need}} & (V) \\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ (\lambda x.x)\ z \mapsto_{\text{need}} & (I) \\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ \boxed{z}\ \mathsf{in}\ x \mapsto_{\text{need}} & (V) \\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \boxed{x} \mapsto_{\text{need}} & (V) \\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \lambda x.x &
\end{array}$$

As in Section 2, we have annotated each reduction step with the name of the corresponding axiom and we have shaded the occurrences of the variables whose values are needed in the course of the reduction. Only the first shaded occurrence of z triggers the reduction of $(\lambda y.y)\ (\lambda x.x)$. The result of this demand-driven reduction is memoized in the let expression that declares z and thus the two other shaded occurrences of z trigger the (V) rule. This let expression is needed as long as z occurs free in its body; thereafter it can be garbage-collected [6].

4 Some exegesis

Definition 2 packs a lot of information. Let us methodically spell it out:

- The axioms are a mouthful, and so in Section 4.1, we identify their underlying structure by stating a grammar for potential redexes.
- In reduction semantics, evaluation is defined as iterated one-step reduction. However, one-step reduction assumes Barendregt's variable convention, i.e., that all bound variables are distinct, but not all the axioms preserve this convention: naive iteration is thus unsound. Rather than subsequently ensuring hygiene as in Garcia et al.'s construction of a lazy abstract machine [19], we restate one axiom in Section 4.2 to make naive iteration hygienic upfront.

- The evaluation contexts are unusual in that they involve terms that are uniquely decomposable into a delimited evaluation context and an identifier. In Section 4.3, we restate their definition to clearly distinguish between ordinary evaluation contexts and delimited evaluation contexts.
- The one-step reduction of a reduction semantics is implicitly structured in three parts: given a non-answer term, (1, decomposition): locate the next potential redex according to the reduction strategy; (2, contraction): if the potential redex is an actual one, i.e., if the non-answer term is not stuck, contract this actual redex as specified by the axioms; and (3, recomposition): fill the surrounding context with the contractum to construct the next term in the reduction sequence.

Based on Sections 4.1, 4.2, and 4.3, we specify decomposition, contraction, and recomposition in Sections 4.4, 4.5, and 4.6. We then formalize one-step reduction in Section 4.7 and evaluation as iterated one-step reduction in Section 4.8.

4.1 Potential redexes

To bring out the underlying structure of the axioms, let us state a grammar for potential redexes:

$$\mathsf{Pot\ Redexes} \ni R ::= A\ T \mid \mathsf{let}\ x\ \mathsf{be}\ A\ \mathsf{in}\ E[x]$$

where $E[x]$ stands for a non-answer term.

The two forms of answers – value and let expression – give rise to an axiom for each production in the grammar of potential redexes: (I) arises from the application of a value to a term; (C) arises from the application of a let expression to a term; and likewise, (V) and (A) arise from the binding of an answer to a variable whose value is needed.

In the present case of the pure λ_{let}-calculus, all potential redexes are actual ones.

4.2 Barendregt's variable convention

The definition of evaluation as iterated one-step reduction assumes Barendregt's variable convention, i.e., that all bound variables are distinct. Indeed the rules (V), (C) and (A) assume the variable convention when they move a term in the scope of a binding. A reduction step involving (V), however, yields a term where the variable convention does not hold, since V is duplicated and it may contain λ-abstractions and therefore bound variables.

There are many ways to ensure hygiene, if not the variable convention, at all times. We choose to allow λ-bound (not let-bound) variables to overlap, and since no reduction can take place inside a λ-abstraction prior to its application, in Rule (I), we lazily rename λ-bound variables as the need arises, using an auxiliary function to rename the formal parameter of a λ-abstraction with a globally fresh name:

$$(I) \quad (\lambda x.T)\ T_1 \to \mathsf{let}\ x'\ \mathsf{be}\ T_1\ \mathsf{in}\ T[x'/x] \quad \text{where } x' \text{ is fresh}$$

This modification preserves Barendregt's variable convention for let-bound variables. We thus assume that any initial term satisfies the convention for let-bound variables (otherwise we appropriately rename it).

Other alternatives exist for ensuring hygiene. We have explored several of them, and in our experience they lead to semantic artifacts that are about as simple and understandable as the ones presented here. The alternative we chose here, i.e., the modification of Rule (I), corresponds to, and is derived into the same renaming side condition as in Maraist, Odersky, and Wadler's natural semantics [26, Figure 11].

4.3 The evaluation contexts

The grammar of contexts for call by need, given in Definition 2, is unusual compared to the one for call by name given in Definition 1. Call-by-need evaluation contexts have an additional constructor involving the term "$E[x]$" for which there exists an identifier x in the eye of a delimited context E. Spelling out the decomposition function (see Section 4.5 and Figure 3) shows that these delimited contexts are constructed *outside in* whereas all the others are constructed *inside out*. Let us make it explicit which are which by adopting an isomorphic representation of contexts as a list of frames:

$$\begin{array}{rl}
\textsf{Context Frames} \ni F & ::= \square\ T \mid \textsf{let}\ x\ \textsf{be}\ \square\ \textsf{in}\ E_{oi}[x] \mid \textsf{let}\ x\ \textsf{be}\ T\ \textsf{in}\ \square \\
\textsf{Outside-in Contexts} \ni E_{oi} & ::= \varepsilon \mid F \circ E_{oi} \\
\textsf{Inside-out Contexts} \ni E_{io} & ::= \varepsilon \mid F \circ E_{io}
\end{array}$$

Here ε is the empty list, $\circ$ is the list constructor, and $\square$ is the hole in a context frame. For example, the context $E = ([\,]\ T_1)\ T_2$ is isomorphic to $E_{io} = (\square\ T_1) \circ (\square\ T_2) \circ \varepsilon$ which is equivalent to $E_{oi} = (\square\ T_2) \circ (\square\ T_1) \circ \varepsilon$ in the sense that, as defined in Section 4.4, $\langle E_{io}, T\rangle_{io} \Uparrow_{\mathrm{rec}} T'$ and $\langle E_{oi}, T\rangle_{oi} \Uparrow_{\mathrm{rec}} T'$ for all T.

NB. As pointed out in Footnote 1 page 2, in this BNF of context frames, the notation "$E_{oi}[x]$" is meant to represent a term that uniquely decomposes into an outside-in evaluation context E_{oi} and a variable x. From Section 5.1 onwards, we take notational advantage of this paired representation to short-cut the subsequent decomposition of this term into E_{oi} and x towards the potential redex contracted in Rule (V) in Section 4.6.

4.4 Recomposition

Outside-in contexts and inside-out contexts are recomposed (or again are 'plugged' or 'filled') as follows:

Definition 3 (recomposition of outside-in contexts). *An outside-in context E_{oi} is recomposed around a term T into a term T' whenever $\langle E_{oi}, T\rangle_{oi} \Uparrow_{\mathrm{rec}} T'$ holds. (See Figure 1.)*

Definition 4 (recomposition of inside-out contexts). *An inside-out context E_{io} is recomposed around a term T into a term T' whenever $\langle E_{io}, T\rangle_{io} \Uparrow_{\mathrm{rec}} T'$ holds. (See Figure 2.)*

$$\frac{}{\langle \varepsilon,\, T\rangle_{oi} \Uparrow_{\text{rec}} T} \qquad \frac{\langle E_{oi},\, T\rangle_{oi} \Uparrow_{\text{rec}} T_1 \qquad \langle E'_{oi},\, x\rangle_{oi} \Uparrow_{\text{rec}} T_2}{\langle(\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E'_{oi}[x]) \circ E_{oi},\, T\rangle_{oi} \Uparrow_{\text{rec}} \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T_2}$$

$$\frac{\langle E_{oi},\, T\rangle_{oi} \Uparrow_{\text{rec}} T_0}{\langle(\Box\ T_1) \circ E_{oi},\, T\rangle_{oi} \Uparrow_{\text{rec}} T_0\ T_1} \qquad \frac{\langle E_{oi},\, T\rangle_{oi} \Uparrow_{\text{rec}} T_2}{\langle(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box) \circ E_{oi},\, T\rangle_{oi} \Uparrow_{\text{rec}} \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T_2}$$

Fig. 1. Recomposition of outside-in contexts

$$\frac{}{\langle \varepsilon,\, T\rangle_{io} \Uparrow_{\text{rec}} T} \qquad \frac{\langle E_{oi},\, x\rangle_{oi} \Uparrow_{\text{rec}} T \qquad \langle E_{io},\, \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T\rangle_{io} \Uparrow_{\text{rec}} T_2}{\langle(\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E_{oi}[x]) \circ E_{io},\, T_1\rangle_{io} \Uparrow_{\text{rec}} T_2}$$

$$\frac{\langle E_{io},\, T_0\ T_1\rangle_{io} \Uparrow_{\text{rec}} T_2}{\langle(\Box\ T_1) \circ E_{io},\, T_0\rangle_{io} \Uparrow_{\text{rec}} T_2} \qquad \frac{\langle E_{io},\, \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T\rangle_{io} \Uparrow_{\text{rec}} T_2}{\langle(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box) \circ E_{io},\, T\rangle_{io} \Uparrow_{\text{rec}} T_2}$$

Fig. 2. Recomposition of inside-out contexts

$$\begin{array}{rcl}
\langle x,\, E_{io}\rangle_{term} & \downarrow_{\text{dec}} & \langle E_{io},\, (\varepsilon,\, x)\rangle_{reroot} \\
\langle \lambda x.T,\, E_{io}\rangle_{term} & \downarrow_{\text{dec}} & \langle E_{io},\, \lambda x.T\rangle_{context} \\
\langle T_0\ T_1,\, E_{io}\rangle_{term} & \downarrow_{\text{dec}} & \langle T_0,\, (\Box\ T_1) \circ E_{io}\rangle_{term} \\
\langle \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T,\, E_{io}\rangle_{term} & \downarrow_{\text{dec}} & \langle T,\, (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box) \circ E_{io}\rangle_{term} \\
\\
\langle \varepsilon,\, A\rangle_{context} & \downarrow_{\text{dec}} & \langle A\rangle_{answer} \\
\langle (\Box\ T_1) \circ E_{io},\, A\rangle_{context} & \downarrow_{\text{dec}} & \langle A\ T_1,\, E_{io}\rangle_{decomposition} \\
\langle (\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E_{oi}[x]) \circ E_{io},\, A\rangle_{context} & \downarrow_{\text{dec}} & \langle \mathsf{let}\ x\ \mathsf{be}\ A\ \mathsf{in}\ E_{oi}[x],\, E_{io}\rangle_{decomposition} \\
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box) \circ E_{io},\, A\rangle_{context} & \downarrow_{\text{dec}} & \langle E_{io},\, \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A\rangle_{context} \\
\\
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box) \circ E_{io},\, (E_{oi},\, x)\rangle_{reroot} & \downarrow_{\text{dec}} & \langle T_1,\, (\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E_{oi}[x]) \circ E_{io}\rangle_{term} \\
\langle F \circ E_{io},\, (E_{oi},\, x)\rangle_{reroot} & \downarrow_{\text{dec}} & \langle E_{io},\, (F \circ E_{oi},\, x)\rangle_{reroot} \\
 & & \text{where } F \neq \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \Box
\end{array}$$

Fig. 3. Decomposition of an answer term into itself
and of a non-answer term into a potential redex and its context

The alert reader will have noticed that each of these recomposition functions, together with the data type of contexts, is in defunctionalized form [13,14,30,31], indicating a large and friendly degree of freedom for implementing a one-step reduction function in a functional programming language.

4.5 Decomposition

Decomposing a non-answer term into a potential redex and its evaluation context is at the heart of a reduction semantics, but outside of the authors' publications, it seems never to be spelled out. Let us do so.

There are many ways to specify decomposition. In our experience, the simplest one is the abstract machine displayed in Figure 3. This machine starts in the configuration $\langle T,\, \varepsilon\rangle_{term}$, for a given term T. It halts in an answer state if the given term contains no potential redex, and in a decomposition state $\langle R,\, E_{io}\rangle_{decomposition}$ otherwise, where R denotes the first potential redex in T and E_{io} its evaluation context according to the reduction strategy specified by the grammar of evaluation contexts.

The *term* and *context* transitions are traditional: one dispatches on a term and the other on the top context frame. The *reroot* transitions locate the let-binder for a variable while maintaining the outside-in context from the binder to its occurrence, zipper-style [21].[4] In effect, the transitions reverse the prefix of an inside-out context into an outside-in context.

4.6 The axioms

In accordance with the new BNF of contexts, the hygienic axioms of Definition 2 are restated as follows:

$$\begin{array}{lrcll}
(I) & (\lambda x.T)\ T_1 & \rightarrow & \mathsf{let}\ x'\ \mathsf{be}\ T_1\ \mathsf{in}\ T[x'/x] & \text{where } x' \text{ is fresh} \\
(V) & \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E_{oi}[x] & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ T & \text{where } \langle E_{oi},\, V\rangle_{oi} \Uparrow_{\text{rec}} T \\
(C) & (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A)\ T_2 & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A\ T_2 & \\
(A) & \mathsf{let}\ x\ \mathsf{be}\ \mathsf{let}\ y\ \mathsf{be}\ T_1 & \rightarrow & \mathsf{let}\ y\ \mathsf{be}\ T_1 & \\
 & \mathsf{in}\ A & & \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ A & \\
 & \mathsf{in}\ E_{oi}[x] & & \quad \mathsf{in}\ E_{oi}[x] &
\end{array}$$

4.7 One-step reduction

The partial function of performing one contraction in a non-answer term is defined as (1) locating a redex and its context through a number of decomposition steps, (2) contracting this redex, and (3) recomposing the resulting contractum into the context:

Definition 5 (one-step reduction). *For any T,*

$$T \mapsto_{\text{need}} T'' \quad \textit{if} \quad \begin{cases} \langle T,\, \varepsilon\rangle_{term} \downarrow^*_{\text{dec}} \langle R,\, E_{io}\rangle_{decomposition} \\ \quad (R,\, T') \;\in\; (I) \cup (V) \cup (C) \cup (A) \\ \langle E_{io},\, T'\rangle_{io} \Uparrow_{\text{rec}} T'' \end{cases}$$

One-step reduction is a partial function because the given term may already be an answer.

4.8 Reduction-based evaluation

Reduction-based evaluation is defined as the iteration of the one-step reduction function. It thus proceeds by enumerating the reduction sequence of any given term:

Definition 6 (reduction-based evaluation). *For any T, $T \mapsto^*_{\text{need}} A$*

Evaluation is a partial function because it may diverge.

[4] Decomposition could be stuck if the initial term contained free variables, but we assume it to be closed.

4.9 Conclusion and perspectives

As illustrated here, there is substantially more than meets the eye in a reduction semantics.

In addition, extensional properties such as unique decomposition, standardization, and hygiene do not only ensure the existence of a deterministic evaluator extensionally, but it is our thesis that they also provide precious intensional guidelines. Indeed, after exegetically spelling out what does not readily meet the eye, things become compellingly simple: refocusing the reduction-based evaluation function immediately gives a reduction-free small-step abstract machine (Section 5.1); fusing its iteration function with its transition functions yields a big-step abstract machine (Section 5.2); compressing the corridor transitions of this abstract machine improves the efficiency of its execution (Section 5.3); refunctionalizing this improved abstract machine with respect to the contexts gives a reduction-free evaluation function in continuation-passing style (Section 5.4); and mapping this evaluation function back to direct style gives a functional implementation of a natural semantics (Section 5.5). All of these semantic artifacts are correct by construction, and their operational behaviors rigorously mirror each other. And should one be tempted to fine-tune one of these semantic artifacts, one is in position to adjust the others to keep their operational behaviors in line, or to understand why this alignment is not possible and where coherence got lost in the fine-tuning [10].

5 Derivation: from small-step reduction semantics to big-step evaluation function

This section implements the programme outlined in Section 4.9.

5.1 Refocusing: from reduction semantics to small-step abstract machine

By recomposing and then immediately decomposing, a reduction-based evaluator takes a detour from a redex site, up to the top of the term, and back down again to the next redex site. Many of the steps that make up this detour can be eliminated by refocusing [15]. Refocusing the reduction-based evaluation function of a reduction semantics yields a reduction-free evaluation function in the form of an abstract machine that directly navigates in a term from redex site to redex site without any detour via the top of the term.

Refocusing replaces successive recompositions and decompositions by a call to a ‘refocus’ function that maps a contractum and its associated (inside-out) evaluation context into a value or a decomposition consisting of the next potential redex and associated evaluation context. Surprisingly, optimal refocusing consists of simply continuing with decomposition from the contractum and its associated evaluation context. This is another reason why we place such store in the decomposition function of a reduction semantics.

The hygienic reduction semantics of Section 4 satisfies the requirements for refocusing [15] and so its reduction-based evaluation function can be mechanically replaced by a reduction-free evaluation function that short-cuts the successive terms in the reduction sequence.

5.2 Lightweight fusion: from small-step abstract machine to big-step abstract machine

The refocused specification of a reduction semantics implements a *small-step* abstract machine. Small-step abstract machines are characterized by a single-step state-transition function which maps a machine configuration to the next and is iterated toward a final state if any. In contrast, big-step abstract machines are characterized by a collection of mutually tail-recursive transition functions mapping a configuration to a final state, if any.

Fusing the composition of a small-step abstract machine's iteration function with its transition functions yields a big-step abstract machine [12]. To this end, we use Ohori and Sasano's lightweight fusion by fixed-point promotion [28].

The difference between the two styles of abstract machines is not typically apparent in the abstract-machine specifications found in programming-language semantics. A machine specification is normally presented as a small-step abstract machine given by reading the transition arrow as the definition of a single-step transition function to be iterated and with the configuration labels as passive components of the configurations. However, the same specification can equally be seen as a big-step abstract machine if the transition labels are interpreted as tail recursive functions, with the transition arrow connecting left- and right-hand sides of their definitions. Ohori and Sasano's correctness-preserving fusion of the small-step machine's transition and iteration functions justifies this dual view.

The difference between the two styles is relevant when we consider the transformation of an abstract machine semantics into an evaluator implementing a natural semantics. Such evaluators are big-step ones, and it is for this reason that we transform small-step machines into big-step machines.

5.3 Transition compression: from big-step abstract machine to big-step abstract machine

Some of the transitions of the abstract machine of Section 5.2 are to intermediate configurations such that the next transition is statically known. These so-called "corridor transitions" can be hereditarily compressed so that the original configuration transitions directly to the final one, skipping all intermediate ones. The resulting abstract machine is displayed in Figure 4.

Proposition 1 (full correctness). *For any T,*

$$T \mapsto^*_{\mathrm{need}} A \;\;\Leftrightarrow\;\; \langle T,\, \varepsilon\rangle_{term} \rightarrow^*_{\mathrm{run}} \langle A\rangle_{answer}.$$

$$
\begin{array}{rcl}
\langle x,\, E_{io}\rangle_{\mathit{term}} & \to_{\mathrm{run}} & \langle E_{io},\, (\varepsilon,\, x)\rangle_{\mathit{reroot}}\\
\langle \lambda x.T,\, E_{io}\rangle_{\mathit{term}} & \to_{\mathrm{run}} & \langle E_{io},\, \lambda x.T\rangle_{\mathit{context}}\\
\langle T_0\ T_1,\, E_{io}\rangle_{\mathit{term}} & \to_{\mathrm{run}} & \langle T_0,\, (\square\ T_1)\circ E_{io}\rangle_{\mathit{term}}\\
\langle \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T,\, E_{io}\rangle_{\mathit{term}} & \to_{\mathrm{run}} & \langle T,\, (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square)\circ E_{io}\rangle_{\mathit{term}}\\[1ex]
\langle \varepsilon,\, A\rangle_{\mathit{context}} & \to_{\mathrm{run}} & \langle A\rangle_{\mathit{answer}}\\
\langle (\square\ T_1)\circ E_{io},\, \lambda x.T\rangle_{\mathit{context}} & \to_{\mathrm{run}} & \langle T[x'/x],\, (\mathsf{let}\ x'\ \mathsf{be}\ T_1\ \mathsf{in}\ \square)\circ E_{io}\rangle_{\mathit{term}}\\
 & & \text{where } x' \text{ is fresh}\\
\langle (\square\ T_2)\circ E_{io},\, \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A\rangle_{\mathit{context}} & \to_{\mathrm{run}} & \langle (\square\ T_2)\circ(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square)\circ E_{io},\, A\rangle_{\mathit{context}}\\
\langle (\mathsf{let}\ x\ \mathsf{be}\ \square\ \mathsf{in}\ E_{oi}[x])\circ E_{io},\, V\rangle_{\mathit{context}} & \to_{\mathrm{run}} & \langle T,\, (\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ \square)\circ E_{io}\rangle_{\mathit{term}}\\
 & & \text{where } \langle E_{oi},\, V\rangle_{oi} \Uparrow_{\mathrm{rec}} T\\
\left\langle \begin{pmatrix}\mathsf{let}\ x\ \mathsf{be}\ \square\\ \mathsf{in}\ E_{oi}[x]\end{pmatrix}\circ E_{io},\, \begin{pmatrix}\mathsf{let}\ y\ \mathsf{be}\ T_1\\ \mathsf{in}\ A\end{pmatrix}\right\rangle_{\mathit{context}} & \to_{\mathrm{run}} & \left\langle \begin{pmatrix}\mathsf{let}\ x\ \mathsf{be}\ \square\\ \mathsf{in}\ E_{oi}[x]\end{pmatrix}\circ\begin{pmatrix}\mathsf{let}\ y\ \mathsf{be}\ T_1\\ \mathsf{in}\ \square\end{pmatrix}\circ E_{io},\, A\right\rangle_{\mathit{context}}\\
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square)\circ E_{io},\, A\rangle_{\mathit{context}} & \to_{\mathrm{run}} & \langle E_{io},\, \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A\rangle_{\mathit{context}}\\[1ex]
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square)\circ E_{io},\, (E_{oi},\, x)\rangle_{\mathit{reroot}} & \to_{\mathrm{run}} & \langle T_1,\, (\mathsf{let}\ x\ \mathsf{be}\ \square\ \mathsf{in}\ E_{oi}[x])\circ E_{io}\rangle_{\mathit{term}}\\
\langle F\circ E_{io},\, (E_{oi},\, x)\rangle_{\mathit{reroot}} & \to_{\mathrm{run}} & \langle E_{io},\, (F\circ E_{oi},\, x)\rangle_{\mathit{reroot}}\\
 & & \text{where } F \neq \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \square
\end{array}
$$

Fig. 4. Abstract machine after transition compression

5.4 Refunctionalization: from abstract machine to continuation-passing interpreter

Reynolds introduced defunctionalization [14, 30] to derive first-order evaluators from higher-order ones. Defunctionalization turns a function type into a sum type, and function application into the application of an apply function dispatching on the sum type. Its left inverse, *refunctionalization* [13], can transform first-order abstract machines into higher-order evaluators. It specifically works on programs that are in defunctionalized form, i.e., in the image of Reynolds's defunctionalization.

The big-step abstract machine of Section 5.3 is not in defunctionalized form with respect to the inside-out reduction contexts. Indeed these contexts are consumed by the two transition functions corresponding to $\langle E_{io},\, A\rangle_{\mathit{context}}$ and $\langle E_{io},\, (E_{oi},\, x)\rangle_{\mathit{reroot}}$ rather than by the single apply function demanded for refunctionalization. This mismatch can be fixed by introducing a sum type discriminating between the (non-context) arguments to the two transition functions and combining them into a single transition function [13]. The left summand (tagged "*ans*") contains an answer, and the right summand (tagged "*ide*") contains a pair of an identifier whose value is needed and an incrementally-constructed outside-in context used to get back to the place in the term where the value was needed.

Three of the context constructors occur on the right-hand sides of their own apply function clauses. When refunctionalized, these correspond to recursive functions and therefore show up as named functions.

The refunctionalized abstract machine is an interpreter for lazy evaluation in continuation-passing style, with the functional representation of the inside-out contexts serving as continuations.

$$\frac{}{x \Downarrow_{\text{eval}} ide(\varepsilon,\, x)} \qquad \frac{}{\lambda x.T \Downarrow_{\text{eval}} ans(\lambda x.T)} \qquad \frac{T_0 \Downarrow_{\text{eval}} r \qquad r\; T_1 \Downarrow_{\text{apply}} r'}{T_0\; T_1 \Downarrow_{\text{eval}} r'}$$

$$\frac{T \Downarrow_{\text{eval}} r \qquad (x,\, T_1,\, r) \Downarrow_{\text{bind}} r'}{\mathsf{let}\; x\; \mathsf{be}\; T_1\; \mathsf{in}\; T \Downarrow_{\text{eval}} r'} \qquad \frac{(ans(A))\; T_2 \Downarrow_{\text{apply}} r \qquad (x,\, T_1,\, r) \Downarrow_{\text{bind}} r'}{(ans(\mathsf{let}\; x\; \mathsf{be}\; T_1\; \mathsf{in}\; A))\; T_2 \Downarrow_{\text{apply}} r'}$$

$$\frac{T[x'/x] \Downarrow_{\text{eval}} r \qquad (x',\, T_1,\, r) \Downarrow_{\text{bind}} r'}{(ans(\lambda x.T))\; T_1 \Downarrow_{\text{apply}} r'}\text{where } x' \text{ is fresh}$$

$$\frac{}{(ide(E_{oi},\, x))\; T_1 \Downarrow_{\text{apply}} ide((\square\; T_1) \circ E_{oi},\, x)}$$

$$\frac{}{(x,\, T_1,\, ans(A)) \Downarrow_{\text{bind}} ans(\mathsf{let}\; x\; \mathsf{be}\; T_1\; \mathsf{in}\; A)} \qquad \frac{T_1 \Downarrow_{\text{eval}} r \qquad (x,\, E_{oi},\, r) \Downarrow_{\text{force}} r'}{(x,\, T_1,\, ide(E_{oi},\, x)) \Downarrow_{\text{bind}} r'}$$

$$\frac{}{(x,\, T_1,\, ide(E_{oi},\, y)) \Downarrow_{\text{bind}} ide((\mathsf{let}\; x\; \mathsf{be}\; T_1\; \mathsf{in}\; \square) \circ E_{oi},\, y)}\text{where } x \neq y$$

$$\frac{\langle E_{oi},\, V\rangle_{oi} \Uparrow_{\text{rec}} T \qquad T \Downarrow_{\text{eval}} r \qquad (x,\, V,\, r) \Downarrow_{\text{bind}} r'}{(x,\, E_{oi},\, ans(V)) \Downarrow_{\text{force}} r'}$$

$$\frac{(x,\, E_{oi},\, ans(A)) \Downarrow_{\text{force}} r \qquad (y,\, T_1,\, r) \Downarrow_{\text{bind}} r'}{(x,\, E_{oi},\, ans(\mathsf{let}\; y\; \mathsf{be}\; T_1\; \mathsf{in}\; A)) \Downarrow_{\text{force}} r'}$$

$$\frac{}{(x,\, E_{oi},\, ide(E'_{oi},\, y)) \Downarrow_{\text{force}} ide((\mathsf{let}\; x\; \mathsf{be}\; \square\; \mathsf{in}\; E_{oi}[x]) \circ E'_{oi},\, y)}$$

Fig. 5. Natural semantics

5.5 Back to direct style: from continuation-passing interpreter to natural semantics

It is a simple matter to transform the continuation-passing interpreter described in Section 5.4 into direct style [8]. The continuations do not represent any control effect other than non-tail calls, so the resulting direct-style interpreter does not require first-class control operators [11].

This interpreter implements a natural semantics (i.e., a big-step operational semantics) for lazy evaluation. This semantics is displayed in Figure 5.

Proposition 2 (full correctness). *For any T,*

$$\langle T,\, \varepsilon\rangle_{term} \rightarrow^*_{\text{run}} \langle A\rangle_{answer} \;\Leftrightarrow\; T \Downarrow_{\text{eval}} ans(A).$$

5.6 Refunctionalization: from natural semantics to higher-order evaluation function

The natural semantics implementation of Section 5.5 is already in defunctionalized form with respect to the first-order outside-in contexts. Indeed, as already mentioned in Section 4.4, the recomposition function of Definition 3 and Figure 1 is the corresponding apply function.

An outside-in context acts as an accumulator recording the path from a variable whose value is needed to its binding site. The recomposition function

$$\begin{aligned}
\mathrm{eval}(x) &= \mathit{ide}(\overline{\lambda} r.r,\, x)\\
\mathrm{eval}(\lambda x.T) &= \mathit{ans}(\lambda x.T)\\
\mathrm{eval}(T_0\ T_1) &= \mathrm{apply}(\mathrm{eval}(T_0), T_1)\\
\mathrm{eval}(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T) &= \mathrm{bind}(x, T_1, \mathrm{eval}(T))\\[1ex]
\mathrm{apply}(\mathit{ans}(\lambda x.T), T_1) &= \mathrm{bind}(x', T_1, \mathrm{eval}(T[x'/x])) \qquad \text{where } x' \text{ is fresh}\\
\mathrm{apply}(\mathit{ans}(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A), T_2) &= \mathrm{bind}(x, T_1, \mathrm{apply}(\mathit{ans}(A), T_2))\\
\mathrm{apply}(\mathit{ide}(h,\, x), T_1) &= \mathit{ide}(\overline{\lambda} r.\mathrm{apply}(h \mathbin{\overline{@}} r, T_1),\, x)\\[1ex]
\mathrm{bind}(x, T_1, \mathit{ans}(A)) &= \mathit{ans}(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A)\\
\mathrm{bind}(x, T_1, \mathit{ide}(h,\, x)) &= \mathrm{force}(x, h, \mathrm{eval}(T_1))\\
\mathrm{bind}(x, T_1, \mathit{ide}(h,\, y)) &= \mathit{ide}(\overline{\lambda} r.\mathrm{bind}(x, T_1, h \mathbin{\overline{@}} r),\, y) \qquad \text{where } x \neq y\\[1ex]
\mathrm{force}(x, h, \mathit{ans}(V)) &= \mathrm{bind}(x, V, h \mathbin{\overline{@}} (\mathit{ans}(V)))\\
\mathrm{force}(x, h, \mathit{ans}(\mathsf{let}\ y\ \mathsf{be}\ T_1\ \mathsf{in}\ A)) &= \mathrm{bind}(y, T_1, \mathrm{force}(x, h, \mathit{ans}(A)))\\
\mathrm{force}(x, h, \mathit{ide}(h',\, y)) &= \mathit{ide}(\overline{\lambda} r.\mathrm{force}(x, h, h' \mathbin{\overline{@}} r),\, y)
\end{aligned}$$

Fig. 6. Higher-order evaluation function

turns this accumulator inside-out again when the variable's value is found. The refunctionalized outside-in contexts are functional representations of these accumulators.

The resulting refunctionalized evaluation function is displayed in Figure 6. Notationally, higher-order functions are introduced with $\overline{\lambda}$ and eliminated with $\overline{@}$, which is infix.

Proposition 3 (full correctness). *For any T,*

$$T \Downarrow_{\mathrm{eval}} \mathit{ans}(A) \quad\Leftrightarrow\quad \mathrm{eval}(T) = \mathit{ans}(A).$$

This higher-order evaluation function exhibits a computational pattern that we find striking because it also occurs in Cartwright and Felleisen's work on extensible denotational language specifications [7]: each valuation function yields either a (left-injected with "*ans*") value or a (right-injected with "*ide*") higher-order function. For each call, this higher-order function may yield another right-injected higher-order function that, when applied, restores this current call. This computational pattern is typical of delimited control: the left inject stands for an expected result, while the right inject acts as an exceptional return that incrementally captures the current continuation. At any rate, this pattern was subsequently re-invented by Fünfrocken to implement process migration [18, 32, 34], and then put to use to implement first-class continuations [25, 29]. In the present case, this pattern embodies two distinct computational aspects—one intensional and the other extensional:

How: The computational pattern is one of delimited control, from the point of use of a let-bound identifier to its point of declaration.

What: The computational effect is one of a write-once state since once the delimited context is captured, it is restored with the value of the let-bound identifier.

These two aspects were instrumental in Cartwright and Felleisen's design of extensible denotational semantics for (undelimited) Control Scheme and for State Scheme [7]. For call by need, this control aspect was recently re-discovered by Garcia, Lumsdaine and Sabry [19], and this store aspect was originally envisioned by Landin [23]. These observations put us in position to write the evaluation function in direct style, either with delimited control operators (one control delimiter for each let declaration, and one control abstraction for each occurrence of a let-declared identifier whose value is needed [3]), or with a state monad. We elaborate this point further in the extended version of this article.

6 Conclusion

Semantics should be call by need.
– Rod Burstall

Over the years, the two key features of lazy evaluation – demand-driven computation and memoization of intermediate results – have elicited a fascinating variety of semantic artifacts, each with its own originality and elegance. It is our overarching thesis that spelling out the methodical search for the next potential redex that is implicit in a reduction semantics paves the way towards other semantic artifacts that not only are uniformly inter-derivable and sound by construction *but also correspond to what one crafts by hand*. Elsewhere, we have already shown that refocusing, etc. do not merely apply to purely syntactic theories such as, e.g., Felleisen and Hieb's syntactic theories of sequential control and state [16,27]: the methodology also applies to call by need with a global heap of memo-thunks [1,5], and to graph reduction, connecting term graph rewriting systems à la Barendregt et al. and graph reduction machines à la Turner [35]. Here, we have shown that the methodology also applies to Ariola et al.'s purely syntactic account of call-by-need.

Acknowledgments: Thanks are due to the anonymous reviewers. We are also grateful to Zena Ariola, Kenichi Asai, Ronald Garcia, Oleg Kiselyov, Kristoffer Rose and Chung-chieh Shan for discussions and comments.

References

1. Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004.
2. Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.
3. Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In Xavier Leroy, editor, *Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages*, pages 64–76, Venice, Italy, January 2004. ACM Press.
4. Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984.

5. Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007.
6. Roel Bloo and Kristoffer Høgsbro Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *CSN-95: Computer Science in the Netherlands*, pages 62–72, 1995.
7. Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In Masami Hagiya and John C. Mitchell, editors, *Proceedings of the 1994 International Symposium on Theoretical Aspects of Computer Software*, number 789 in Lecture Notes in Computer Science, pages 244–272, Sendai, Japan, April 1994. Springer.
8. Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.
9. Olivier Danvy. Defunctionalized interpreters for programming languages. In Peter Thiemann, editor, *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 131–142, Victoria, British Columbia, September 2008. ACM Press. Invited talk.
10. Olivier Danvy. From reduction-based to reduction-free normalization. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced Functional Programming, Sixth International School*, number 5382 in Lecture Notes in Computer Science, pages 66–164, Nijmegen, The Netherlands, May 2008. Springer. Lecture notes including 70+ exercises.
11. Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 299–310, San Francisco, California, June 1992. ACM Press.
12. Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Information Processing Letters*, 106(3):100–109, 2008.
13. Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8):534–549, 2009.
14. Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press.
15. Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, Aarhus, Denmark, November 2004.
16. Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.
17. Daniel P. Friedman, Abdulaziz Ghuloum, Jeremy G. Siek, and Lynn Winebarger. Improving the lazy Krivine machine. *Higher-Order and Symbolic Computation*, 20(3):271–293, 2007.
18. Stefan Fünfrocken. Transparent migration of Java-based mobile agents. In Kurt Rothermel and Fritz Hohl, editors, *Mobile Agents, Second International Workshop, MA'98, Proceedings*, volume 1477 of *Lecture Notes in Computer Science*, pages 26–37, Stuttgart, Germany, September 1998. Springer.
19. Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. In Benjamin C. Pierce, editor, *Proceedings of the Thirty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 153–164, Savannah, GA, January 2009. ACM Press.

20. Peter Henderson and James H. Morris Jr. A lazy evaluator. In Susan L. Graham, editor, *Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 95–103. ACM Press, January 1976.
21. Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
22. Mark B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68:105–111, 1989.
23. Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
24. John Launchbury. A natural semantics for lazy evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, January 1993. ACM Press.
25. Florian Loitsch. *Scheme to JavaScript Compilation.* PhD thesis, Université de Nice, Nice, France, March 2009.
26. John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.
27. Johan Munk. A study of syntactic and semantic artifacts and its application to lambda definability, strong normalization, and weak normalization in the presence of state. Master's thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, May 2007. BRICS research report RS-08-3.
28. Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In Matthias Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 143–154, Nice, France, January 2007. ACM Press.
29. Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In Benjamin Pierce, editor, *Proceedings of the 2005 ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 216–227, Tallinn, Estonia, September 2005. ACM Press.
30. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363-397, 1998, with a foreword [31].
31. John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.
32. Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. In Paolo Ciancarini and Alexander L. Wolf, editors, *Coordination Languages and Models, Third International Conference, COORDINATION '99, Proceedings*, volume 1594 of *Lecture Notes in Computer Science*, pages 211–226, Amsterdam, The Netherlands, April 1999. Springer.
33. Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.
34. Wei Tao. *A portable mechanism for thread persistence and migration.* PhD thesis, University of Utah, Salt Lake City, Utah, 2001.
35. Ian Zerny. On graph rewriting, reduction and evaluation. In Zoltán Horváth, Viktória Zsók, Peter Achten, and Pieter Koopman, editors, *Trends in Functional Programming Volume 10*, Komárno, Slovakia, June 2009. Intellect Books. To appear.

To be presented at FLOPS 2010.