A Synthetic Operational Account of Call-by-Need Evaluation

Olivier Danvy
Department of Computer Science, Aarhus University
danvy@cs.au.dk

Ian Zerny [*]
Department of Computer Science, Aarhus University [†]
ian@zerny.dk

Abstract

We present the first operational account of call by need that connects syntactic theory and implementation practice. Syntactic theory: the storeless operational semantics using syntax rewriting to account for demand-driven computation and for caching intermediate results. Implementational practice: the store-based operational technique using memo-thunks to implement demand-driven computation and to cache intermediate results for subsequent sharing. The implementational practice was initiated by Landin and Wadsworth and is prevalent today to implement lazy programming languages such as Haskell. The syntactic theory was initiated by Ariola, Felleisen, Maraist, Odersky and Wadler and is prevalent today to reason equationally about lazy programs, on par with Barendregt et al.'s term graphs. Nobody knows, however, how the theory of call by need compares to the practice of call by need: all that is known is that the theory of call by need agrees with the theory of call by name, and that the practice of call by need optimizes the practice of call by name.

Our operational account takes the form of three new calculi for lazy evaluation of lambda-terms and our synthesis takes the form of three lock-step equivalences. The first calculus is a hereditarily compressed variant of Ariola et al.'s call-by-need lambda-calculus and makes "neededness" syntactically explicit. The second calculus distinguishes between strict bindings (which are induced by demand-driven computation) and non-strict bindings (which are used for caching intermediate results). The third calculus uses memo-thunks and an algebraic store. The first calculus syntactically corresponds to a storeless abstract machine, the second to an abstract machine with local stores, and the third to a lazy Krivine machine, i.e., a traditional store-based abstract machine implementing lazy evaluation. The machines are intensionally compatible with extensional reasoning about lazy programs and they are lock-step equivalent. Each machine functionally corresponds to a natural semantics for call by need in the style of Launchbury, though for non-preprocessed λ-terms.

Our results reveal a genuine and principled unity of computational theory and computational practice, one that readily applies to variations on the general theme of call by need.

[*] Recipient of the Google Europe Fellowship in Programming Technology. This research is supported in part by this Google Fellowship.

[†] New affiliation as of June 2013: Google

PPDP '13, September 16-18 2013, Madrid, Spain. Copyright is held by the owner/author(s). Publication rights licensed to ACM.
Copyright © ACM 978-1-4503-2154-9/13/09 . . . $15.00.
http://dx.doi.org/10.1145/2505879.2505884.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; F.1.1 [*Computation by Abstract Devices*]: Models of Computation; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics

General Terms languages, theory

Keywords abstract machines, call by need, functional correspondence, lazy evaluation, natural semantics, reduction semantics, syntactic correspondence, syntactic theories

1. Introduction

Seen in the historical HAKMEM report [11, Item 101B]:

```
Let x be a continued fraction
  p0 + q0/(p1 + q1/(...)) = p0 + q0/x'
where x' is again a continued fraction
and the p's and q's are integers. [...]
Instead of a list of p's and q's,
let x be a subroutine
producing its next p and q each time it is called.
Thus on its first usage, x will "output" p0 and q0
and, in effect, change itself into x'.
```

Lovely example of a lazy list and of its evaluation, isn't it? And to think it was programmed in assembly language too...

But what is lazy evaluation? Lazy evaluation is an embodiment of *computation on demand* and of *memoization of intermediate results* for subsequent reuse, in case of subsequent similar demands. In the λ-calculus, lazy evaluation improves the standard reduction of λ-terms. In functional programming languages, lazy evaluation is implemented by passing actual parameters "by need" both to user-defined functions and to data constructors. Call by need is traditionally implemented with memo-thunks, as in Gosper's procedural representation of continued fractions above: parameterless procedures that delay computation and which, when forced, memoize their result in the store. And indeed lazy abstract machines canonically use a store to manage memo-thunks. Alternatively, they use updateable graphs rather than abstract-syntax trees, in which case these graphs play the rôle of the store [22, 23]. Today, the best-known such store-based abstract machine is probably Peyton Jones's Spineless Tagless G-Machine [38], which is the run-time system of the Glasgow Haskell compiler.

Does this mean that a store is inherently necessary to account for lazy evaluation? Perhaps surprisingly, the answer is no: in the mid-1990's [6], in a simultaneous tour de force, Ariola and Felleisen [5] and Maraist, Odersky and Wadler [34] provided a purely syntactic, storeless operational account of call by need.

So what has happened since? Somewhat unexpectedly, the store-based implementation technique of using memo-thunks and the storeless operational account of call by need have remained disconnected.[1] Instead, for example, the control aspect of the storeless operational account has most recently been sought to emulate this storage effect using delimited continuations [28] or more generally a computational monad [19].

So what else has happened instead? Sestoft [43] and Maraist et al. [34] have continued to investigate store-based natural semantics for lazy evaluation, following Launchbury's [32]. Recently, Nakata and Hasegawa have shown the equivalence of variants of the calculus and Launchbury's natural semantics [36] and Chang and Felleisen have designed a new calculus with one axiom and shown it to correspond with Launchbury's natural semantics [14]. Other lazy abstract machines have been designed as well [26, 28]. All of these results provide extensional equivalence between the semantics, which yield equivalent values on identical terminating terms. Also, some of the machines are ostensibly derived but none of the derivation methods seem to have been subsequently reused.

Does this mean that the long-emerged investigation of lazy evaluation is still an ever-expanding and disconnected exploratory process? Actually, no. Ager et al. [3] have inter-derived a store-based call-by-need evaluator and a lazy version of the Krivine machine with a store (Figure 2, page 3), using the functional correspondence between evaluators and abstract machines summarized in Appendix A.2 and originally due to Reynolds [2, 41]. Biernacka and Danvy [13, Section 9] have inter-derived a reduction semantics of a λ-calculus with explicit substitutions and a store, $\lambda\widehat{\rho}_1$, and the *same* lazy version of the Krivine machine with a store (Figure 2), using the syntactic correspondence between reduction semantics and abstract machines summarized in Appendix A.1. Pirog and Biernacki [39] have inter-derived Launchbury and Sestoft's natural semantics for lazy evaluation and Peyton Jones's Spineless Tagless G-Machine, using Reynolds's functional correspondence and formalizing this correspondence in Coq. We have inter-derived combinatory graph reduction over term graphs à la Barendregt et al. [10] and Turner's graph-reduction machine [45], including the Y combinator [21]. And Ariola, Downen, Herbelin, Nakata and Saurin have put this inter-derivational unity of semantic artifacts to use for call-by-need sequent calculi [4].

So what is the contribution of the present article? We report a similar major unified progress in the investigation of lazy evaluation in the λ-calculus. Macroscopically, as depicted in Figure 13, page 10, we present a connection between the two major accounts of lazy evaluation—store-based and storeless—across the three major styles of operational semantics: reduction semantics, abstract machines and natural semantics. And microscopically, as defined in Section 4, our connection is based on a *lock-step equivalence* between the small-step semantics.

In what does our synthetic account differ from others? Our account is dual to Hardin, Maranget and Pagano's [29], who seek invariant structures in existing abstract machines using a calculus of explicit substitutions, and to Douence and Fradet's [23], who seek common structures in existing abstract machines to establish a taxonomy. Indeed we *end* with abstract machines as semantic artifacts that are in the common range of the syntactic correspondence (Appendix A.1) and of the functional correspondence (Appendix A.2). We also illustrate how tuning one semantics mechanically gives rise to another, e.g., Launchbury's natural semantics and Sestoft's abstract machine.

Keeping in mind the answers to the series of questions above, the rest of this article is organized as follows:[2] Section 2 first contrasts call by name and call by need. Section 3 presents our first starting point: a store-based lazy version of the Krivine abstract machine. Section 4 presents our notion of lock-step equivalence. Section 5 presents our second starting point: a reduction semantics for call-by-need reduction. Section 6 then presents abstract machines for call-by-need evaluation, and Section 7 natural semantics for call-by-need evaluation. Figure 1 depicts our overall story.

Prerequisites and notations: We expect the reader to be acquainted with the formats of a (small-step) reduction semantics, an abstract machine, and a (big-step) natural semantics, as can be gathered, e.g., in Felleisen et al.'s recent textbook [25] and in Danvy's lecture notes at AFP 2008 [17]. We use x and y to range over names and use subscripts as well as primes to distinguish meta-variables in a syntactic category, e.g., x_0, x_1, x', x''. We assume all initial terms to be closed λ-terms defined by the usual BNF of the pure λ-calculus:

$$\Lambda \ni t ::= x \mid \lambda x.t \mid t\ t$$

and let $fv(t)$ denote the set of free variables in t. When unambiguous, we allow meta-variables from different semantic specifications to overlap. When ambiguous, we superscript the meta-variables with the notion of reduction of the particular semantics, e.g., $t^{\mathcal{R}}$.

2. Call by need vs. call by name

Lazy evaluation is an optimization of the standard reduction of λ-terms and as such we expect any two strategies for lazy evaluation to assign identical meanings to identical λ-terms. However, the operational behaviors specified by the two strategies can be readily observed on even small programs, such as this fully applied expression using Church numbers:

$$\overbrace{c_m\ (c_m\ (\cdots (c_m}^{n}\ id\ id) \cdots)\ id)\ id$$

where $c_n = \lambda s.\lambda z.\overbrace{s\ (s\ \cdots (s}^{n}\ z) \cdots)$. Under call by name, the above expression takes an exponential number of steps to reduce, whereas under call by need, the number of steps is polynomial.

In this regard, call-by-need evaluation is a particular optimization and it is this optimization we want to characterize (so we are not interested in optimal reduction here [8, 33]). As language users and implementors alike, we are interested in reasoning about the time and space complexity of programs under call by need. In other words, *we are concerned with the operational behavior of its specification as opposed to the equational theory it enables*, which is already aptly covered by call by name. We therefore wish to precisely relate the operational behavior specified by the call-by-need λ-calculus to the established method of implementing the call-by-need evaluation strategy. To this end, we first argue for what is a canonical specification of the call-by-need reduction strategy (Section 3). We then define a notion of lock-step equivalence by which we can prove the operational behavior of reduction strategies equivalent up to a fine-grained notion of steps (Section 4).

3. Store-based call-by-need evaluation and reduction

The call-by-need evaluation strategy originates with Wadsworth [46] and was subsequently and independently used for programming

[1] Indeed, the only known property of Ariola et al.'s call-by-need λ-calculus is its completeness with respect to the standard reduction of the λ-calculus (i.e., call by name).

[2] In the wake of the previous 700 papers about lazy evaluation, it is something of a challenge to write an original introduction for yet another article about lazy evaluation. In the 701st paper [19], Danvy et al. started with an unashamedly apocryphal anecdote illustrating computation on demand and memoization of intermediate results. In the present paper, and as no doubt spotted by the zealous reader already, we *wrote* the present introduction in call-by-need style with a series of Socratic questions concluded with a request to keep their answers in mind.

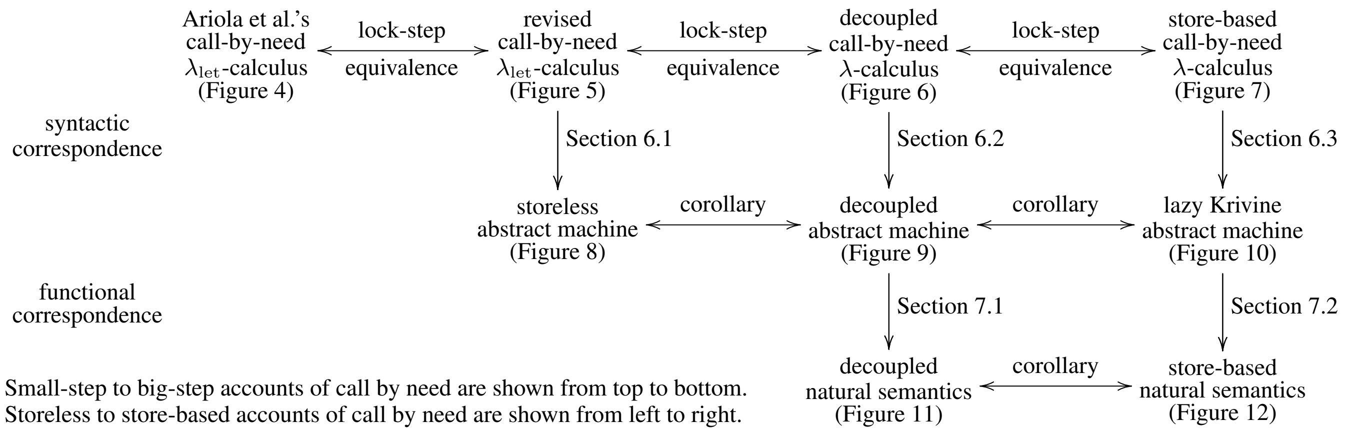

Ariola et al.'s call-by-need λ_{let}-calculus (Figure 4)
lock-step equivalence
revised call-by-need λ_{let}-calculus (Figure 5)
lock-step equivalence
decoupled call-by-need λ-calculus (Figure 6)
lock-step equivalence
store-based call-by-need λ-calculus (Figure 7)
syntactic correspondence
Section 6.1
Section 6.2
Section 6.3
storeless abstract machine (Figure 8)
corollary
decoupled abstract machine (Figure 9)
corollary
lazy Krivine abstract machine (Figure 10)
functional correspondence
Section 7.1
Section 7.2
decoupled natural semantics (Figure 11)
corollary
store-based natural semantics (Figure 12)
Small-step to big-step accounts of call by need are shown from top to bottom.
Storeless to store-based accounts of call by need are shown from left to right.

Figure 1: Local picture: semantics for call-by-need evaluation

Syntax:

$$\begin{array}{rcl}
\mathsf{Term} \ni t & ::= & x \mid \lambda x.t \mid t\,t \\
\mathsf{Value} \ni v & ::= & \lambda x.t \\
\mathsf{Context} \ni E & ::= & \square \mid E\,t \mid x := E \\
\mathsf{Store} \ni \sigma & ::= & \varepsilon \mid \sigma[x = t]
\end{array}$$

Transition rules:

$$\begin{array}{rcl}
\langle \lambda x.t,\, E,\, \sigma\rangle_{term} & \to_{\mathcal{S}} & \langle E,\, \lambda x.t,\, \sigma\rangle_{cont} \\
\langle t_0\, t_1,\, E,\, \sigma\rangle_{term} & \to_{\mathcal{S}} & \langle t_0,\, E[\square\, t_1],\, \sigma\rangle_{term} \\
\langle x,\, E,\, \sigma\rangle_{term} & \to_{\mathcal{S}} & \langle t,\, E[x := \square],\, \sigma\rangle_{term} \\
 & & \text{where } t = \sigma(x) \\
 & & \quad \text{and } t \notin \mathsf{Value} \\
\langle x,\, E,\, \sigma\rangle_{term} & \to_{\mathcal{S}} & \langle E,\, v,\, \sigma\rangle_{cont} \\
 & & \text{where } v = \sigma(x) \\
 & & \\
\langle \square,\, v,\, \sigma\rangle_{cont} & \to_{\mathcal{S}} & \langle v,\, \sigma\rangle_{ans} \\
\langle E[\square\, t_1],\, \lambda x.t,\, \sigma\rangle_{cont} & \to_{\mathcal{S}} & \langle t[x'/x],\, E,\, \sigma[x' = t_1]\rangle_{term} \\
 & & \text{where } x' \notin \mathrm{dom}(\sigma) \\
\langle E[x := \square],\, v,\, \sigma\rangle_{cont} & \to_{\mathcal{S}} & \langle E,\, v,\, \sigma[x = v]\rangle_{cont}
\end{array}$$

Execution starts in a *term*-configuration with an empty context and an empty store, and proceeds through successive transitions. In the second *cont*-transition, an actual parameter is delayed in a thunk. In the second-to-last *term*-transition, a thunk is forced. In the last *cont*-transition, a thunk has completed and its result is memoized.

Figure 2: Lazy version of the Krivine abstract machine

Syntax:

$$\begin{array}{rcl}
\mathsf{Term} \ni t & ::= & x \mid \lambda x.t \mid t\,t \mid x := t \\
\mathsf{Value} \ni v & ::= & \lambda x.t \\
\mathsf{Context} \ni E & ::= & \square \mid E\,t \mid x := E \\
\mathsf{Store} \ni \sigma & ::= & \varepsilon \mid \sigma[x = t] \\
\mathsf{Redex} \ni r & ::= & v\,t \mid x := v \mid x
\end{array}$$

Contraction rules:

$$\begin{array}{lrcl}
(I) & \langle E[(\lambda x.t)\, t_1],\, \sigma\rangle & \to & \langle E[t[x'/x]],\, \sigma[x' = t_1]\rangle \\
 & & & \text{where } x' \notin \mathrm{dom}(\sigma) \\
(V) & \langle E[x := v],\, \sigma\rangle & \to & \langle E[v],\, \sigma[x = v]\rangle \\
(L) & \langle E[x],\, \sigma\rangle & \to & \langle E[x := t],\, \sigma\rangle \\
 & & & \text{where } t = \sigma(x) \\
 & & & \quad \text{and } t \notin \mathsf{Value} \\
(\widetilde{V}) & \langle E[x],\, \sigma\rangle & \to & \langle E[v],\, \sigma\rangle \\
 & & & \text{where } v = \sigma(x)
\end{array}$$

$$\mathcal{S} = (I) \cup (V) \cup (L) \cup (\widetilde{V})$$

Standard one-step reduction:

$$\langle t,\, \sigma\rangle \mapsto_{\mathcal{S}} \langle t',\, \sigma'\rangle \text{ iff } \begin{cases} \text{decomposition:} & t \;=\; E[r] \\ (\langle E[r],\, \sigma\rangle,\, \langle E[t''],\, \sigma'\rangle) \in \mathcal{S} & \\ \text{recomposition:} & t' \;=\; E[t''] \end{cases}$$

Figure 3: The reduction semantics corresponding to Figure 2

languages by Henderson and Morris [30] and by Friedman and Wise [27]. All of these specifications have one idea in common: to delay the evaluation of actual parameters with thunks allocated in a global store, and to force this evaluation on demand and memoize the resulting value in the thunk. In this section, we specify a canonical machine for call-by-need evaluation of pure λ-terms which reflects this implementational practice. We then inter-derive the corresponding notion of one-step reduction.

3.1 A machine for call-by-need evaluation

Our starting point is the properly tail-recursive and lazy variant of the Krivine machine displayed in Figure 2 [16]. This machine features memo-thunks in a global store. In words: Terms are pure λ-terms. Values are λ-abstractions. Evaluation contexts consist of the empty context, an application context, and an update context. The machine uses two transition functions: *term* dispatching on terms and *cont* dispatching on evaluation contexts. Here demand-driven computation is implemented by the third *term*-transition and the second *cont*-transition, while memoization is implemented by the last *cont*-transition.

This machine represents a canonical implementation of the call-by-need evaluation strategy using actual substitutions. The update contexts represent "update markers" in the sense of Fairbairn and Wray's Three Instruction Machine [24]. Furthermore, this machine can be inter-derived with traditional store-based call-by-need evaluators, as shown by Ager et al. [3].

Definition 1 (reduction-free evaluation)**.** *A term $t \in \Lambda$ evaluates to a value v iff*

$$\langle t,\, \square,\, \varepsilon\rangle_{term} \to^{*}_{\mathcal{S}} \langle v,\, \sigma\rangle_{ans}$$

holds, where $\to^{*}_{\mathcal{S}}$ is the transitive closure of $\to_{\mathcal{S}}$. (See Figure 2.)

3.2 A semantics for call-by-need reduction

Using Biernacka and Danvy's syntactic correspondence between reduction semantics and abstract machines [13], we can inter-derive the lazy Krivine machine of Section 3.1 with a reduction-based counterpart. This reduction-based semantics takes the form of a reduction semantics and is displayed in Figure 3. Compared to the machine, an update context is given a term representation in the form of an update expression. The grammar of potential redexes gives rise to the four contraction rules. Here, demand-driven computation is captured by Rule (I) and Rule (L), and memoization is captured by Rule (V).

This reduction semantics is the closure-free counterpart of $\lambda\widehat{\rho}_{l}$ [13, Section 9].

Definition 2 (reduction-based evaluation)**.** *A term $t \in \Lambda$ reduces to a value v iff $\langle t,\, \varepsilon\rangle \mapsto^*_{\mathcal{S}} \langle v,\, \sigma\rangle$ holds, where $\mapsto^*_{\mathcal{S}}$ is the reflexive-transitive closure of $\mapsto_{\mathcal{S}}$. (See Figure 3.)*

3.3 From evaluation to reduction

In summary, we have defined a reduction semantics for call-by-need reduction. Through the syntactic correspondence, this semantics corresponds to a store-based lazy version of the Krivine abstract machine which is inter-derivable with a traditional store-based lazy evaluator. Therefore, these semantics specify the canonical evaluation strategy of call by need.

Proposition 3 (full correctness)**.** *For any term $t \in \Lambda$,*

$$\langle t,\, \square,\, \varepsilon\rangle_{term} \rightarrow^*_{\mathcal{S}} \langle v_1,\, \sigma_1\rangle_{ans} \iff \langle t,\, \varepsilon\rangle \mapsto^*_{\mathcal{S}} \langle v_2,\, \sigma_2\rangle$$

where $v_1 =_\alpha v_2$ and $\sigma_1 =_\alpha \sigma_2$.

Proof. Corollary of the full correctness of refocusing [20, 44]. □

4. Lock-step equivalence

Our notion of lock-step equivalence is based on Milner's weak bisimulation or observable equivalence of processes [35, Chapter 5]. In contrast to bisimulation where one equates objects within the same system, we are here interested in relating objects between separately defined systems. Also, we are not concerned with labeling.

We understand a process to be the steps defined by a transition system, in our case the standard reduction sequence as defined by the standard one-step reduction of a reduction semantics. Two processes are lock-step equivalent if any step taken by one can be mirrored by the other modulo the steps considered internal to the other process:

Definition 4 (Lock-step equivalence)**.** *Let $\rightarrow_\alpha$ be a transition system with internal transitions $\rightarrow_{\hat{\alpha}}$ and let $\rightarrow_\beta$ be a transition system with internal transitions $\rightarrow_{\hat{\beta}}$. A binary relation $\mathbf{R}$ between the states of $\rightarrow_\alpha$ and $\rightarrow_\beta$ is a lock-step relation if for all $a\,\mathbf{R}\,b$:*

$$a \rightarrow_\alpha a' \Rightarrow \exists b' :\ b \rightarrow^*_{\hat{\beta}} \rightarrow_\beta \rightarrow^*_{\hat{\beta}} b' \ \wedge\ a'\,\mathbf{R}\,b'$$

and

$$b \rightarrow_\beta b' \Rightarrow \exists a' :\ a \rightarrow^*_{\hat{\alpha}} \rightarrow_\alpha \rightarrow^*_{\hat{\alpha}} a' \ \wedge\ a'\,\mathbf{R}\,b'$$

Two states a and b are lock-step equivalent, $a \,{}_\alpha{\approx}_\beta\, b$, iff there exists a lock-step relation $\mathbf{R}$ such that $a\,\mathbf{R}\,b$. If both sets of internal transitions are empty, then the lock-step relation is akin to a strong bisimulation.

5. Reduction semantics for call-by-need reduction

This section presents four reduction semantics for call-by-need evaluation together with their standard reduction. The first one is the λ_{let}-calculus and is due to Ariola, Felleisen, Maraist, Odersky,

Syntax:

$$\begin{array}{rcl}
\text{Term} \ni t & ::= & x \mid \lambda x.t \mid t\,t \mid \text{let}\ x = t\ \text{in}\ t \\
\text{Value} \ni v & ::= & \lambda x.t \\
\text{Answer} \ni a & ::= & \lambda x.t \mid \text{let}\ x = t\ \text{in}\ a \\
\text{Context} \ni E & ::= & \square \mid E\,t \mid \text{let}\ x = t\ \text{in}\ E \mid \text{let}\ x = E\ \text{in}\ E[x] \\
\text{Redex} \ni r & ::= & a\,t \mid \text{let}\ x = a\ \text{in}\ E[x]
\end{array}$$

Contraction rules:

$$\begin{array}{llcl}
(I) & (\lambda x.t)\ t_1 & \rightarrow & \text{let}\ x = t_1\ \text{in}\ t \\
(C) & (\text{let}\ x = t_1\ \text{in}\ a)\ t_2 & \rightarrow & \text{let}\ x = t_1\ \text{in}\ a\ t_2 \\
(V) & \text{let}\ x = v\ \text{in}\ E[x] & \rightarrow & \text{let}\ x = v\ \text{in}\ E[v] \\
(A) & \text{let}\ x = \text{let}\ y = t_1 & \rightarrow & \text{let}\ y = t_1 \\
 & \qquad\quad \text{in}\ a & & \quad \text{in let}\ x = a \\
 & \quad \text{in}\ E[x] & & \qquad \text{in}\ E[x]
\end{array}$$

$$\mathcal{R} = (I) \cup (C) \cup (V) \cup (A)$$

Standard one-step reduction:

$$t \mapsto_{\mathcal{R}} t' \text{ iff } \begin{cases} \text{decomposition:} & t = E[r] \\ \text{contraction:} & (r,\, t'') \in \mathcal{R} \\ \text{recomposition:} & t' = E[t''] \end{cases}$$

Figure 4: The call-by-need λ_{let}-calculus

and Wadler (Section 5.1). We then calculate a revised semantics where (1) we hereditarily contract potential redexes, and (2) we make explicit when denotables are needed from their point of use to their point of declaration (Section 5.2). This latter distinction leads us to introducing a new term to represent this def-use chain: a strict let expression. A more uniform distinction between strict and non-strict let expressions leads us to 'decoupling' the reduction semantics into one with two contexts (Section 5.3). Interpreting one of these two contexts as a global store yields the reduction semantics of Figure 3 (see Section 5.4) which specifies the canonical evaluation strategy of call by need: Ariola et al.'s reduction semantics therefore also specifies the canonical evaluation strategy of call by need.

5.1 Storeless reduction semantics

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 [5, 6, 34], renaming non-terminals for notational uniformity. This calculus and its standard reduction are displayed in Figure 4. In words: Terms are pure λ-terms with let expressions declaring denotables. Values are λ-abstractions. Answers are let expressions nested around a value. Evaluation contexts are terms with a single hole and are constructed according to the standard call-by-need reduction strategy. Redexes come in two forms: the application of an answer, or the binding of an answer to a denotable whose value is needed. Each gives rise to a pair of contraction rules:

- Rules (I) and (C) arise from the application of an answer. (I) introduces a let binding, while (C) allows let bindings to commute with applications.
- Rules (V) and (A) arise from the binding of an answer to a variable whose value is needed. (V) hygienically substitutes a definiens (here: a value) for a variable occurrence, while (A) re-associates let bindings.

The standard one-step reduction is the compatible closure over the contraction rules with respect to the evaluation contexts.

Definition 5 (reduction-based evaluation [6]). *A term $t \in \Lambda$ reduces to an answer a iff $t \mapsto^*_{\mathcal{R}} a$ holds, where $\mapsto^*_{\mathcal{R}}$ is the reflexive-transitive closure of $\mapsto_{\mathcal{R}}$. (See Figure 4.)*

5.2 Revised storeless reduction semantics

In this section, we develop a revised version of the storeless reduction semantics with contraction rules that more closely match the rules of the store-based reduction semantics in Section 3.2.

Hereditary contraction: Examining the contraction rules of Figure 4, we see that a contractum of Rule (C) contains a redex of the form $a\ t$. This redex can thus be further contracted by either Rule (C) or Rule (I). Likewise, a contractum of Rule (A) contains a redex of the form $\mathsf{let}\ x = a\ \mathsf{in}\ E[x]$. This redex can thus be further contracted by either Rule (A) or Rule (V). More precisely, by straightforward induction on the following definition of answer contexts,

$$\mathsf{Ans\ Ctx} \ni A ::= \Box \mid \mathsf{let}\ x = t\ \mathsf{in}\ A$$

we relate the following terms under the reflexive transitive closure of the standard one-step reduction relation:

$$\begin{array}{rl} A[\lambda x.t]\ t_1 \mapsto^*_{\mathcal{R}} & A[\mathsf{let}\ x = t_1\ \mathsf{in}\ t] \\ \mathsf{let}\ x = A[v]\ \mathsf{in}\ E[x] \mapsto^*_{\mathcal{R}} & A[\mathsf{let}\ x = v\ \mathsf{in}\ E[v]] \end{array}$$

Strict let expressions: The reduction semantics of Section 5.1 cleverly specifies what it means for a denotable to be "needed." Specifically, a denotable x is needed in a term t if t can be uniquely decomposed into $E[x]$. Therefore, the let expression "$\mathsf{let}\ x = t_1\ \mathsf{in}\ t$" can be in one of two states: if $t = E[x]$, we say that the let is *strict*, forcing evaluation of the definiens; and if $t \neq E[x]$, we say that the let is non-strict, postponing the evaluation of the definiens.

Let us make this property syntactically explicit in terms, using a strict let expression "$\mathsf{let}\ x := t\ \mathsf{in}\ E[x]$" where x is needed in the body $E[x]$. In contrast, the original let expression "$\mathsf{let}\ x = t\ \mathsf{in}\ t$" is a non-strict let expression. Strict let expressions are then introduced and eliminated with the following rules:

$$\begin{array}{l} \mathsf{let}\ x = t_1\ \mathsf{in}\ E[x] \to \mathsf{let}\ x := t_1\ \mathsf{in}\ E[x] \\ \mathsf{let}\ x := v\ \mathsf{in}\ E[x] \to \mathsf{let}\ x = v\ \mathsf{in}\ E[v] \end{array}$$

In the case where t_1 is a value, this introduction rule and this elimination rule are applied consecutively in the reduction sequence. To cater for that case, we fuse these two rules into a new one, $(\widetilde{V})$.

Applying the two changes (hereditary contraction and introduction of strict let expressions) to the reduction semantics of Section 5.1 we obtain the semantics displayed in Figure 5.

Definition 6 (reduction-based evaluation). *A term $t \in \Lambda$ reduces to a value v in an answer context A iff $t \mapsto^*_{\mathcal{C}} A[v]$ holds, where $\mapsto^*_{\mathcal{C}}$ is the reflexive-transitive closure of $\mapsto_{\mathcal{C}}$. (See Figure 5.)*

Proposition 7 (full correctness). *For any closed $t \in \Lambda$, $t\ {}_{\mathcal{R}}{\approx}_{\mathcal{C}}\ t$.*

Proof. There exists a lock-step relation over $\mathcal{R}$ and $\mathcal{C}$, where (C), (A) and (L) are internal transitions. (See Figure 14.) □

Syntax:

$$\begin{array}{rl} \mathsf{Term} \ni t & ::= x \mid \lambda x.t \mid t\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \mid \mathsf{let}\ x := t\ \mathsf{in}\ E[x] \\ \mathsf{Value} \ni v & ::= \lambda x.t \\ \mathsf{Ans\ Ctx} \ni A & ::= \Box \mid \mathsf{let}\ x = t\ \mathsf{in}\ A \\ \mathsf{Context} \ni E & ::= \Box \mid E\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ E \mid \mathsf{let}\ x := E\ \mathsf{in}\ E[x] \\ \mathsf{Redex} \ni r & ::= A[v]\ t \mid \mathsf{let}\ x := A[v]\ \mathsf{in}\ E[x] \mid \mathsf{let}\ x = t\ \mathsf{in}\ E[x] \end{array}$$

Contraction rules:

$$\begin{array}{lrl} (I) & A[\lambda x.t]\ t_1 \to & A[\mathsf{let}\ x = t_1\ \mathsf{in}\ t] \\ (V) & \mathsf{let}\ x := A[v]\ \mathsf{in}\ E[x] \to & A[\mathsf{let}\ x = v\ \mathsf{in}\ E[v]] \\ (L) & \mathsf{let}\ x = t\ \mathsf{in}\ E[x] \to & \mathsf{let}\ x := t\ \mathsf{in}\ E[x] \\ & & \text{where } t \notin \mathsf{Value} \\ (\widetilde{V}) & \mathsf{let}\ x = v\ \mathsf{in}\ E[x] \to & \mathsf{let}\ x = v\ \mathsf{in}\ E[v] \end{array}$$

$$\mathcal{C} = (I) \cup (V) \cup (L) \cup (\widetilde{V})$$

Standard one-step reduction:

$$t \mapsto_{\mathcal{C}} t' \text{ iff } \begin{cases} \text{decomposition:} & t = E[r] \\ \text{contraction:} & (r, t'') \in \mathcal{C} \\ \text{recomposition:} & t' = E[t''] \end{cases}$$

Figure 5: The revised call-by-need λ_{let}-calculus

Syntax:

$$\begin{array}{rl} \mathsf{Term} \ni t & ::= x \mid \lambda x.t \mid t\ t \mid \mathsf{let}\ x := t\ \mathsf{in}\ A \\ \mathsf{Value} \ni v & ::= \lambda x.t \\ \mathsf{Non\text{-}strict\ Context} \ni A & ::= \Box \mid \mathsf{let}\ x = t\ \mathsf{in}\ A \\ \mathsf{Strict\ Context} \ni E & ::= \Box \mid E\ t \mid \mathsf{let}\ x := E\ \mathsf{in}\ A \\ \mathsf{Redex} \ni r & ::= v\ t \mid \mathsf{let}\ x := v\ \mathsf{in}\ A \mid x \end{array}$$

Contraction rules:

$$\begin{array}{lrl} (I) & \langle E[(\lambda x.t)\ t_1],\ A\rangle \to & \langle E[t],\ A[\mathsf{let}\ x = t_1\ \mathsf{in}\ \Box]\rangle \\ (V) & \langle E[\mathsf{let}\ x := v\ \mathsf{in}\ A_1],\ A\rangle \to & \langle E[v],\ A[\mathsf{let}\ x = v\ \mathsf{in}\ A_1]\rangle \\ (L) & \langle E[x],\ A\rangle \to & \langle E[\mathsf{let}\ x := t\ \mathsf{in}\ A_2],\ A_1\rangle \\ & & \text{where } A = A_1[\mathsf{let}\ x = t\ \mathsf{in}\ A_2] \\ & & \text{and } t \notin \mathsf{Value} \\ (\widetilde{V}) & \langle E[x],\ A\rangle \to & \langle E[v],\ A\rangle \\ & & \text{where } A = A_1[\mathsf{let}\ x = v\ \mathsf{in}\ A_2] \end{array}$$

$$\mathcal{D} = (I) \cup (V) \cup (L) \cup (\widetilde{V})$$

Standard one-step reduction:

$$\langle t,\ A\rangle \mapsto_{\mathcal{D}} \langle t',\ A'\rangle \text{ iff } \begin{cases} \text{decomposition:} & t = E[r] \\ (\langle E[r],\ A\rangle,\ \langle E[t''],\ A'\rangle) \in \mathcal{D} & \\ \text{recomposition:} & t' = E[t''] \end{cases}$$

Figure 6: The decoupled call-by-need λ-calculus

5.3 Decoupled reduction semantics

The reduction semantics of Section 5.1 distinguishes strict and non-strict forms only in its specification of evaluation contexts. In Section 5.2, strictness is made explicit in terms, yet strict and non-strict forms remain coupled during contraction. Examining the contraction rules of Figure 5, we see that strict let expressions are introduced to guide computation while non-strict let expressions are introduced to store intermediate results. In this section, we decouple strict and non-strict contexts, thereby separating the concerns of computation from the concerns of mere storage of intermediate results. The resulting semantics is displayed in Figure 6. The defining difference is that non-strict let expressions (in the form of non-strict contexts) are no longer interleaved in the terms. Therefore:

- Terms no longer include non-strict let expressions and strict let expressions delimit just the non-strict context.
- Strict contexts are terms with a hole in strict position: the operand of an application, or the term bound by a strict let expression.
- Non-strict contexts are nested non-strict let expressions.
- Rules (I) and (V) directly place non-strict let bindings in the non-strict context.

Syntax:

$$\begin{array}{rl} \mathsf{Term} \ni t & ::= x \mid \lambda x.t \mid t\,t \mid x := t \\ \mathsf{Value} \ni v & ::= \lambda x.t \\ \mathsf{Context} \ni E & ::= \square \mid E\,t \mid x := E \\ \mathsf{Store} \ni \sigma & ::= \varepsilon \mid \sigma[x = t] \\ \mathsf{Redex} \ni r & ::= v\,t \mid x := v \mid x \end{array}$$

Contraction rules:

$$\begin{array}{ll} (I) & \langle E[(\lambda x.t)\,t_1],\,\sigma\rangle \to \langle E[t],\,\sigma[x = t_1]\rangle \\ (V) & \langle E[x := v],\,\sigma\rangle \to \langle E[v],\,\sigma[x = v]\rangle \\ (L) & \langle E[x],\,\sigma\rangle \to \langle E[x := t],\,\sigma\rangle \\ & \quad \text{where } t = \sigma(x) \\ & \quad \text{and } t \notin \mathsf{Value} \\ (\widetilde{V}) & \langle E[x],\,\sigma\rangle \to \langle E[v],\,\sigma\rangle \\ & \quad \text{where } v = \sigma(x) \end{array}$$

$$\mathcal{S} = (I) \cup (V) \cup (L) \cup (\widetilde{V})$$

Standard one-step reduction:

$$\langle t,\,\sigma\rangle \mapsto_{\mathcal{S}} \langle t',\,\sigma'\rangle \text{ iff } \begin{cases} \text{decomposition:} & t\ = E[r] \\ (\langle E[r],\,\sigma\rangle,\,\langle E[t''],\,\sigma'\rangle) \in \mathcal{S} & \\ \text{recomposition:} & t'\ = E[t''] \end{cases}$$

Figure 7: The store-based call-by-need λ-calculus

- Rules (L) and $(\widetilde{V})$ directly look up denotables in the non-strict context.

The syntax of a strict let expression now consists of a non-strict context. This context arises from Rule (L) where the name x is needed and has a binding to t in the non-strict context A. This context is split in two: A_1 contains bindings that are lexically visible to t and becomes the new non-strict context, thus making the bindings available during evaluation of t; while A_2 contains bindings of names that are not lexically visible to t and is stored in the strict let expression and subsequently restored once evaluation of t completes in Rule (V).

The standard one-step reduction is now defined over a term and a non-strict context:

Definition 8 (reduction-based evaluation)**.** *A term $t \in \Lambda$ reduces to a value v and a non-strict context A iff $\langle t,\,\square\rangle \mapsto^*_{\mathcal{D}} \langle v,\,A\rangle$ holds, where $\mapsto^*_{\mathcal{D}}$ is the reflexive-transitive closure of $\mapsto_{\mathcal{D}}$. (See Figure 6.)*

Proposition 9 (full correctness)**.** *For any closed $t \in \Lambda$,*

$$t \;{}_{\mathcal{C}}{\approx}_{\mathcal{D}}\; \langle t,\,\square\rangle$$

Proof. There exists a lock-step relation over $\mathcal{C}$ and $\mathcal{D}$, with no internal transitions.
(See Figure 15.) $\square$

5.4 Store-based reduction semantics

The reduction semantics of Section 5.3 distinguishes between strict and non-strict contexts—strict contexts for guiding computation and non-strict contexts for storing intermediate results. In this section, we accentuate this distinction by interpreting strict contexts as evaluation contexts and by representing non-strict contexts—which, by hygiene, all declare distinct denotables—with a global store. The result is a syntactic theory of the traditional implementation technique for lazy evaluation, the one using memo-thunks in a global store. This semantics is displayed in Figure 7. The defining difference is that the global store is never delimited. Therefore:

- Undelimiting update expressions (contexts) replace the delimiting strict let expressions (contexts) in the decoupled semantics.

The standard one-step reduction is defined over a term and a store:

Definition 10 (reduction-based evaluation)**.** *A term $t \in \Lambda$ reduces to a value v with a store σ iff $\langle t,\,\varepsilon\rangle \mapsto^*_{\mathcal{S}} \langle v,\,\sigma\rangle$ holds, where $\mapsto^*_{\mathcal{S}}$ is the reflexive-transitive closure of $\mapsto_{\mathcal{S}}$. (See Figure 7.)*

Proposition 11 (full correctness)**.** *For any closed $t \in \Lambda$,*

$$\langle t,\,\square\rangle \;{}_{\mathcal{D}}{\approx}_{\mathcal{S}}\; \langle t,\,\varepsilon\rangle$$

Proof. There exists a lock-step relation over $\mathcal{D}$ and $\mathcal{S}$, with no internal transitions.
(See Figure 16.) $\square$

5.5 Summary and conclusions

We have presented four lock-step equivalent reduction semantics for call by need:

Corollary 12 (full correctness)**.** *For any closed $t \in \Lambda$,*

$$t \;{}_{\mathcal{R}}{\approx}_{\mathcal{C}}\; t \;{}_{\mathcal{C}}{\approx}_{\mathcal{D}}\; \langle t,\,\square\rangle \;{}_{\mathcal{D}}{\approx}_{\mathcal{S}}\; \langle t,\,\varepsilon\rangle$$

Each of these reduction semantics captures a descriptive aspect of call by need: the first one, which is due to Ariola et al., is storeless and the fourth one is store-based and accounts for the traditional implementation technique of using memo-thunks: Figure 7 is the implicitly hygienic version of Figure 3, which syntactically corresponds to the lazy Krivine machine of Figure 2.

6. Abstract machines for call-by-need evaluation

In this section, we derive abstract machines from each of the reduction semantics presented in Sections 5.2, 5.3, and 5.4. To this end, we use the syntactic correspondence developed by Biernacka and Danvy [13]. The method consists of a series of program transformations between a reduction-based evaluation function, as typically specified by a reduction semantics, and a reduction-free evaluation function, as typically specified by an abstract machine (see Appendix A.1). We do not display the abstract machine corresponding to the reduction semantics presented in Section 5.1 (Ariola et al.'s) because it has already been derived by Danvy et al. [19, Figure 4].

Following common practice, the reduction semantics of Section 5 implicitly assume hygiene in the contraction rules. However, when specifying an abstract machine as the basis for an implementation, the method of ensuring hygiene should be explicit. Mirroring implementation practice, we thread a stream of fresh names with the reduction sequence:

$$X \in \mathsf{FreshNames} = \nu X.\mathsf{Name} \times X.$$

Each contraction rule therefore inherits and synthesizes this stream.

Following Danvy et al.'s hygiene strategy [19, Section 4.2], we choose to rename λ-bound names when introducing let expressions, thereby enforcing that all let-bound names are distinct. To this end, we modify the (I)-rule of each reduction semantics:

$$\begin{array}{ll} (I)^{\mathcal{C}} & \langle E[(\lambda x.t)\,t_1],\,(x',\,X)\rangle \to \langle E[\mathsf{let}\ x' = t_1\ \mathsf{in}\ t[x'/x]],\,X\rangle \\ (I)^{\mathcal{D}} & \langle E[(\lambda x.t)\,t_1],\,A,\,(x',\,X)\rangle \to \langle E[t[x'/x]],\,A[\mathsf{let}\ x' = t_1\ \mathsf{in}\ \square],\,X\rangle \\ (I)^{\mathcal{S}} & \langle E[(\lambda x.t)\,t_1],\,\sigma,\,(x',\,X)\rangle \to \langle E[t[x'/x]],\,\sigma[x' = t_1],\,X\rangle \end{array}$$

For each of our reduction semantics, the syntactic correspondence mechanically yields a reduction-free abstract machine where intermediate steps in the reduction sequence have been deforested away.

6.1 Storeless abstract machine

Figure 8 displays the abstract machine derived from the storeless reduction semantics of Section 5.2. It uses the same definition of terms, values, answer contexts and evaluation contexts. (See Figure 5.)

$$\begin{array}{rcll}
\langle \lambda x.t,\ E\rangle_{term} & \xrightarrow{X;X}_{\mathcal{C}} & \langle E,\ \lambda x.t\rangle_{cont} & \\
\langle t_0\ t_1,\ E\rangle_{term} & \xrightarrow{X;X}_{\mathcal{C}} & \langle t_0,\ E[\Box\ t_1]\rangle_{term} & \\
\langle \mathsf{let}\ x = t_1\ \mathsf{in}\ t,\ E\rangle_{term} & \xrightarrow{X;X}_{\mathcal{C}} & \langle t,\ E[\mathsf{let}\ x = t_1\ \mathsf{in}\ \Box]\rangle_{term} & \\
\langle \mathsf{let}\ x := t\ \mathsf{in}\ E_1[x],\ E\rangle_{term} & \xrightarrow{X;X}_{\mathcal{C}} & \langle t,\ E[\mathsf{let}\ x := \Box\ \mathsf{in}\ E_1[x]]\rangle_{term} & \\
\langle x,\ E\rangle_{term} & \xrightarrow{X;X}_{\mathcal{C}} & \langle t,\ E_1[\mathsf{let}\ x := \Box\ \mathsf{in}\ E_2[x]]\rangle_{term} & \text{where } E = E_1[\mathsf{let}\ x = t\ \mathsf{in}\ E_2] \text{ and } t \notin \mathsf{Value} \\
\langle x,\ E\rangle_{term} & \xrightarrow{X;X}_{\mathcal{C}} & \langle E,\ v\rangle_{cont} & \text{where } E = E_1[\mathsf{let}\ x = v\ \mathsf{in}\ E_2] \\
\\
\langle \Box,\ A[v]\rangle_{cont} & \xrightarrow{X;X}_{\mathcal{C}} & \langle A[v]\rangle_{ans} & \\
\langle E[\Box\ t_1],\ A[\lambda x.t]\rangle_{cont} & \xrightarrow{(x',X);X}_{\mathcal{C}} & \langle t[x'/x],\ E[A[\mathsf{let}\ x' = t_1\ \mathsf{in}\ \Box]]\rangle_{term} & \\
\langle E[\mathsf{let}\ x = t_1\ \mathsf{in}\ \Box],\ A[v]\rangle_{cont} & \xrightarrow{X;X}_{\mathcal{C}} & \langle E,\ \mathsf{let}\ x = t_1\ \mathsf{in}\ A[v]\rangle_{cont} & \\
\langle E[\mathsf{let}\ x := \Box\ \mathsf{in}\ E_1[x]],\ A[v]\rangle_{cont} & \xrightarrow{X;X}_{\mathcal{C}} & \langle E[A[\mathsf{let}\ x = v\ \mathsf{in}\ E_1]],\ v\rangle_{cont} &
\end{array}$$

Execution starts in a $term$-configuration with an empty context and proceeds through successive transitions. The stream of fresh names X is threaded through. In the second $cont$-transition, an actual parameter is delayed in a non-strict let expression. In the second-to-last $term$-transition, a non-strict let expression is replaced by a strict let expression, thereby forcing the evaluation of its definiens. In the last $cont$-transition, the evaluation of the definiens has completed and the strict let expression is replaced by a non-strict let expression declaring the resulting value.

Figure 8: Storeless machine for call by need

We note that the abstract machine of Figure 8 is essentially the same as the abstract machines of Garcia et al. [28] and Danvy et al. [19], which differ only with respect to their handling of hygiene. This equivalence arises from two facts: (1) the hereditary compression in the revised semantics is superseded by transition compression of the abstract machine, and (2) even without introducing strict let expressions, their context counterpart must still be represented to guide evaluation in the abstract machine.

Definition 13 (reduction-free evaluation). *A term $t \in \Lambda$ evaluates to a value in an answer context $A[v]$ iff*

$$\langle t,\ \Box\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathcal{C}} \langle A[v]\rangle_{ans}$$

holds, where $\rightarrow^{*}_{\mathcal{C}}$ is the transitive closure of $\rightarrow_{\mathcal{C}}$. (See Figure 8.) Notationally we use $\xrightarrow{X;X'}{}^{*}_{\mathcal{C}}$ to express that X is the input stream and X′ is a suffix of X obtained after iterating $\rightarrow_{\mathcal{C}}$.

NB. Assuming the initial term to be a pure λ-term (i.e., to contain no let expressions), we can omit the third and fourth $term$-transitions. Starting from a pure λ-term, the forms $\mathsf{let}\ x = t\ \mathsf{in}\ t$ and $\mathsf{let}\ x := t\ \mathsf{in}\ E[x]$ are indeed never constructed in the course of execution.

Proposition 14 (full correctness). *For any term $t \in \Lambda$,*

$$t \mapsto^{*}_{\mathcal{C}} A_1[v_1] \iff \langle t,\ \Box\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathcal{C}} \langle A_2[v_2]\rangle_{ans}$$

where $v_1 =_\alpha v_2$. (In fact, $v_1 = v_2$ if, in the reduction sequence, we pick fresh names according to the stream of fresh names threaded in the abstract machine.)

6.2 Decoupled abstract machine

Figure 9 displays the abstract machine derived from the decoupled reduction semantics of Section 5.3. It uses the same definition of terms, values, strict contexts and non-strict contexts. (See Figure 6.)

Definition 15 (reduction-free evaluation). *A term $t \in \Lambda$ evaluates to a value v with a non-strict context A iff*

$$\langle t,\ \Box,\ \Box\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathcal{D}} \langle v,\ A\rangle_{ans}$$

holds, where $\rightarrow^{*}_{\mathcal{D}}$ is the transitive closure of $\rightarrow_{\mathcal{D}}$. (See Figure 9.)

As in Section 6.1, assuming the initial term to be a pure λ-term, we can omit the third $term$-transition.

$$\begin{array}{rcl}
\langle \lambda x.t,\ E,\ A\rangle_{term} & \xrightarrow{X;X}_{\mathcal{D}} & \langle E,\ \lambda x.t,\ A\rangle_{cont} \\
\langle t_0\ t_1,\ E,\ A\rangle_{term} & \xrightarrow{X;X}_{\mathcal{D}} & \langle t_0,\ E[\Box\ t_1],\ A\rangle_{term} \\
\langle \mathsf{let}\ x := t\ \mathsf{in}\ A_1,\ E,\ A\rangle_{term} & \xrightarrow{X;X}_{\mathcal{D}} & \langle t,\ E[\mathsf{let}\ x := \Box\ \mathsf{in}\ A_1],\ A\rangle_{term} \\
\langle x,\ E,\ A\rangle_{term} & \xrightarrow{X;X}_{\mathcal{D}} & \langle t,\ E[\mathsf{let}\ x := \Box\ \mathsf{in}\ A_2],\ A_1\rangle_{term} \\
 & & \text{where } A = A_1[\mathsf{let}\ x = t\ \mathsf{in}\ A_2] \\
 & & \quad \text{and } t \notin \mathsf{Value} \\
\langle x,\ E,\ A\rangle_{term} & \xrightarrow{X;X}_{\mathcal{D}} & \langle E,\ v,\ A\rangle_{cont} \\
 & & \text{where } A = A_1[\mathsf{let}\ x = v\ \mathsf{in}\ A_2] \\
\\
\langle \Box,\ v,\ A\rangle_{cont} & \xrightarrow{X;X}_{\mathcal{D}} & \langle v,\ A\rangle_{ans} \\
\langle E[\Box\ t_1],\ \lambda x.t,\ A\rangle_{cont} & \xrightarrow{(x',X);X}_{\mathcal{D}} & \langle t[x'/x],\ E,\ A[\mathsf{let}\ x' = t_1\ \mathsf{in}\ \Box]\rangle_{term} \\
\langle E[\mathsf{let}\ x := \Box,\ v,\ A\rangle_{cont} & \xrightarrow{X;X}_{\mathcal{D}} & \langle E,\ v,\ A[\mathsf{let}\ x = v\ \mathsf{in}\ A_1]\rangle_{cont} \\
\quad \mathsf{in}\ A_1[x]] & &
\end{array}$$

Execution starts in a $term$-configuration with two empty contexts and proceeds through successive transitions. The stream of fresh names X is threaded through. In the second $cont$-transition, an actual parameter is delayed in a non-strict context. In the second-to-last $term$-transition, a non-strict let expression is replaced by a strict let expression, thereby forcing the evaluation of its definiens. In the last $cont$-transition, the evaluation of the definiens has completed and the strict let expression is replaced back by a non-strict let expression declaring the resulting value.

Figure 9: Decoupled machine for call by need

Proposition 16 (full correctness). *For any term $t \in \Lambda$,*

$$\langle t,\ \Box\rangle \mapsto^{*}_{\mathcal{D}} \langle v_1,\ A_1\rangle \iff \langle t,\ \Box,\ \Box\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\mathcal{D}} \langle v_2,\ A_2\rangle_{ans}$$

where $v_1 =_\alpha v_2$. (Again, $v_1 = v_2$ if, in the reduction sequence, we pick fresh names according to the stream of fresh names threaded in the abstract machine.)

6.3 Store-based abstract machine

Figure 10 displays the abstract machine derived from the store-based reduction semantics of Section 5.4. It uses the same definition of terms, values, evaluation contexts and stores. (See Figure 7.)

$$
\begin{array}{rcll}
\langle \lambda x.t,\, E,\, \sigma\rangle_{term} & \xrightarrow{X;X}_{\mathcal{S}} & \langle E,\, \lambda x.t,\, \sigma\rangle_{cont} & \\
\langle t_0\, t_1,\, E,\, \sigma\rangle_{term} & \xrightarrow{X;X}_{\mathcal{S}} & \langle t_0,\, E[\square\, t_1],\, \sigma\rangle_{term} & \\
\langle x := t,\, E,\, \sigma\rangle_{term} & \xrightarrow{X;X}_{\mathcal{S}} & \langle t,\, E[x := \square],\, \sigma\rangle_{term} & \\
\langle x,\, E,\, \sigma\rangle_{term} & \xrightarrow{X;X}_{\mathcal{S}} & \langle t,\, E[x := \square],\, \sigma\rangle_{term} & \\
 & & \text{where } t = \sigma(x) & \\
 & & \quad\text{and } t \notin \mathsf{Value} & \\
\langle x,\, E,\, \sigma\rangle_{term} & \xrightarrow{X;X}_{\mathcal{S}} & \langle E,\, v,\, \sigma\rangle_{cont} & \\
 & & \text{where } v = \sigma(x) & \\
 & & & \\
\langle \square,\, v,\, \sigma\rangle_{cont} & \xrightarrow{X;X}_{\mathcal{S}} & \langle v,\, \sigma\rangle_{ans} & \\
\langle E[\square\, t_1],\, \lambda x.t,\, \sigma\rangle_{cont} & \xrightarrow{(x',\, X);X}_{\mathcal{S}} & \langle t[x'/x],\, E,\, \sigma[x' = t_1]\rangle_{term} & \\
\langle E[x := \square],\, v,\, \sigma\rangle_{cont} & \xrightarrow{X;X}_{\mathcal{S}} & \langle E,\, v,\, \sigma[x = v]\rangle_{cont} &
\end{array}
$$

Execution starts in a *term*-configuration with an empty context and an empty store, and proceeds through successive transitions. The store σ and the stream of fresh names X are threaded through. In the second *cont*-transition, an actual parameter is delayed in a thunk. In the second-to-last *term*-transition, a thunk is forced. In the last *cont*-transition, a thunk has completed and its result is memoized.

Figure 10: Store-based machine for call by need

$$\frac{}{\langle \lambda x.t,\, A\rangle \;{}^{X}\!\Downarrow^{X}_{\mathcal{D}}\; \langle \lambda x.t,\, A\rangle}$$

$$\frac{\langle t_0,\, A\rangle \;{}^{X}\!\Downarrow^{(x',\, X')}_{\mathcal{D}}\; \langle \lambda x.t,\, A'\rangle \quad \langle t[x'/x],\, A'[\mathsf{let}\ x' = t_1\ \mathsf{in}\ \square]\rangle \;{}^{X'}\!\Downarrow^{X''}_{\mathcal{D}}\; \langle v,\, A''\rangle}{\langle t_0\, t_1,\, A\rangle \;{}^{X}\!\Downarrow^{X''}_{\mathcal{D}}\; \langle v,\, A''\rangle}$$

$$\frac{\langle t,\, A\rangle \;{}^{X}\!\Downarrow^{X'}_{\mathcal{D}}\; \langle v,\, A'\rangle}{\langle \mathsf{let}\ x := t\ \mathsf{in}\ A_1[x],\, A\rangle \;{}^{X}\!\Downarrow^{X'}_{\mathcal{D}}\; \langle v,\, A'[\mathsf{let}\ x = v\ \mathsf{in}\ A_1]\rangle}$$

$$\frac{A = A_1[\mathsf{let}\ x = t\ \mathsf{in}\ A_2] \quad \langle t,\, A_1\rangle \;{}^{X}\!\Downarrow^{X'}_{\mathcal{D}}\; \langle v,\, A_1'\rangle}{\langle x,\, A\rangle \;{}^{X}\!\Downarrow^{X'}_{\mathcal{D}}\; \langle v,\, A_1'[\mathsf{let}\ x = v\ \mathsf{in}\ A_2]\rangle}\ \text{where } t \notin \mathsf{Value}$$

$$\frac{A = A_1[\mathsf{let}\ x = v\ \mathsf{in}\ A_2]}{\langle x,\, A\rangle \;{}^{X}\!\Downarrow^{X'}_{\mathcal{D}}\; \langle v,\, A_1[\mathsf{let}\ x = v\ \mathsf{in}\ A_2]\rangle}$$

Figure 11: Decoupled natural semantics for call by need

Definition 17 (reduction-free evaluation)**.** *A term $t \in \Lambda$ evaluates to a value v with a store σ iff*

$$\langle t,\, \square,\, \varepsilon\rangle_{term} \xrightarrow{X;X'\,*}_{\mathcal{S}} \langle v,\, \sigma\rangle_{ans}$$

holds, where $\rightarrow^*_{\mathcal{S}}$ is the transitive closure of $\rightarrow_{\mathcal{S}}$. (See Figure 10.)

As in Sections 6.1 and 6.2, assuming the initial term to be a pure λ-term, we can omit the third *term*-transition. The abstract machine then coincides with the lazy Krivine machine in Figure 2.

Proposition 18 (full correctness)**.** *For any term $t \in \Lambda$,*

$$\langle t,\, \varepsilon\rangle \mapsto^*_{\mathcal{S}} \langle v_1,\, \sigma_1\rangle \iff \langle t,\, \square,\, \varepsilon\rangle_{term} \xrightarrow{X;X'\,*}_{\mathcal{S}} \langle v_2,\, \sigma_2\rangle_{ans}$$

where $v_1 =_\alpha v_2$. (One more time, $v_1 = v_2$ if, in the reduction sequence, we pick fresh names according to the stream of fresh names threaded in the abstract machine.)

6.4 Summary and conclusions

We have presented three equivalent abstract machines for call by need:

Corollary 19 (full correctness)**.** *For any $t \in \Lambda$,*

$$
\begin{array}{cl}
 & \langle t,\, \square\rangle_{term} \xrightarrow{X;X'\,*}_{\mathcal{C}} \langle A_1[v_1]\rangle_{ans} \\
\Updownarrow & \langle t,\, \square,\, \square\rangle_{term} \xrightarrow{X;X'\,*}_{\mathcal{D}} \langle v_2,\, A_2\rangle_{ans} \\
\Updownarrow & \langle t,\, \square,\, \varepsilon\rangle_{term} \xrightarrow{X;X'\,*}_{\mathcal{S}} \langle v_3,\, \sigma\rangle_{ans}
\end{array}
$$

where $v_1 = v_2 = v_3$.

Each of these abstract machines captures a descriptive aspect of call by need: the two first ones are storeless and the third one is store-based and accounts for the traditional implementation technique of using memo-thunks.

7. Natural semantics for call-by-need evaluation

In this section, we derive natural semantics from each of the abstract machines derived in Sections 6.2 and 6.3. To this end, we use the functional correspondence initiated by Reynolds [41] and developed by Danvy et al. [2, 3, 17]. The method consists of a series of program transformations between an abstract machine and a natural semantics, as typically specified by a recursive evaluation function in direct style (see Appendix A.2). We do not display the natural semantics corresponding to the storeless abstract machine of Section 6.1 because it has already been derived by Danvy et al. [19, Figure 8].

7.1 Decoupled natural semantics

Figure 11 displays the natural semantics derived from the abstract machine of Section 6.2. It uses the same definition of terms, values and non-strict contexts. (See Figure 6.)

We note that the natural semantics of Figure 11 is essentially the same as Nakata and Hasegawa's instrumented natural semantics [36, Figure 7]. The most notable—and yet superficial—difference is that Nakata and Hasegawa retain the entire structure of the evaluation context as part of their structured heaps whereas we only maintain the structure of the non-strict let expressions.

Definition 20 (evaluation)**.** *A term $t \in \Lambda$ evaluates to a value v with a non-strict context A iff $\langle t,\, \square\rangle \;{}^{X}\!\Downarrow^{X'}_{\mathcal{D}}\; \langle v,\, A\rangle$ holds. (See Figure 11.)*

NB. Assuming the initial term to be a pure λ-term (i.e., to contain no let expressions), we can omit the third rule. Indeed, starting from a pure λ-term, the form let $x := t$ in $A[x]$ can never be constructed by a derivation.

Proposition 21 (full correctness)**.** *For any term $t \in \Lambda$,*

$$\langle t,\, \square,\, \square\rangle_{term} \xrightarrow{X;X'\,*}_{\mathcal{D}} \langle v,\, A\rangle_{ans} \iff \langle t,\, \square\rangle \;{}^{X}\!\Downarrow^{X'}_{\mathcal{D}}\; \langle v,\, A\rangle.$$

7.2 Store-based natural semantics

Figure 12 displays the natural semantics derived from the abstract machine of Section 10. It uses the same definition of terms, values and stores. (See Figure 7.)

Compared to Launchbury's [32] and to Maraist et al.'s [34], the natural semantics in Figure 11 explicitly handles name hygiene. Compared to Sestoft's [43], its handling of name hygiene reflects implementational practice. Also, both Launchbury and Sestoft use preprocessed terms, which prevents a direct syntactic comparison.

Definition 22 (evaluation)**.** *A term $t \in \Lambda$ evaluates to a value v with a store σ iff $\langle t,\, \varepsilon\rangle \;{}^{X}\!\Downarrow^{X'}_{\mathcal{S}}\; \langle v,\, \sigma\rangle$ holds. (See Figure 12.)*

As in Section 7.1, assuming the initial term to be a pure λ-term, we can omit the third rule.

Proposition 23 (full correctness)**.** *For any term $t \in \Lambda$,*

$$\langle t,\, \square,\, \varepsilon\rangle_{term} \xrightarrow{X;X'\,*}_{\mathcal{S}} \langle v,\, \sigma\rangle_{ans} \iff \langle t,\, \varepsilon\rangle \;{}^{X}\!\Downarrow^{X'}_{\mathcal{S}}\; \langle v,\, \sigma\rangle.$$

$$\frac{}{\langle \lambda x.t,\, \sigma\rangle \ {}^{X}{\Downarrow}^{X}_{\mathcal{S}}\ \langle \lambda x.t,\, \sigma\rangle}$$

$$\frac{\langle t_0,\, \sigma\rangle \ {}^{X}{\Downarrow}^{(x',X')}_{\mathcal{S}}\ \langle \lambda x.t,\, \sigma'\rangle \quad \langle t[x'/x],\, \sigma'[x'=t_1]\rangle \ {}^{X'}{\Downarrow}^{X''}_{\mathcal{S}}\ \langle v,\, \sigma''\rangle}{\langle t_0\, t_1,\, \sigma\rangle \ {}^{X}{\Downarrow}^{X''}_{\mathcal{S}}\ \langle v,\, \sigma''\rangle}$$

$$\frac{\langle t,\, \sigma\rangle \ {}^{X}{\Downarrow}^{X'}_{\mathcal{S}}\ \langle v,\, \sigma'\rangle}{\langle x := t,\, \sigma\rangle \ {}^{X}{\Downarrow}^{X'}_{\mathcal{S}}\ \langle v,\, \sigma'[x=v]\rangle}$$

$$\frac{t = \sigma(x) \quad \langle t,\, \sigma\rangle \ {}^{X}{\Downarrow}^{X'}_{\mathcal{S}}\ \langle v,\, \sigma'\rangle}{\langle x,\, \sigma\rangle \ {}^{X}{\Downarrow}^{X'}_{\mathcal{S}}\ \langle v,\, \sigma'[x=v]\rangle}\text{where } t \notin \mathsf{Value}$$

$$\frac{v = \sigma(x)}{\langle x,\, \sigma\rangle \ {}^{X}{\Downarrow}^{X}_{\mathcal{S}}\ \langle v,\, \sigma\rangle}$$

Figure 12: Store-based natural semantics for call by need

7.3 Summary and conclusions

We have presented two equivalent natural semantics for call-by-need evaluation:

Corollary 24 (full correctness)**.** *For any term $t \in \Lambda$,*

$$\langle t,\, \square\rangle \ {}^{X}{\Downarrow}^{X'}_{\mathcal{D}}\ \langle v_1,\, A\rangle \iff \langle t,\, \varepsilon\rangle \ {}^{X}{\Downarrow}^{X'}_{\mathcal{S}}\ \langle v_2,\, \sigma\rangle$$

where $v_1 =_\alpha v_2$.

Each of these natural semantics captures a descriptive aspect of call by need: the first one is storeless and the second one is store-based and accounts for the traditional implementation technique of using memo-thunks.

Perhaps surprisingly, Launchbury's natural semantics is to be found *between* the decoupled and store-based natural semantics presented here. The store-based semantics differs since it uses a global store whereas Launchbury's semantics does not. The decoupled semantics more closely connects with Launchbury's semantics. The fourth rule of Figure 11 mirrors Launchbury's *Variable* rule in that the binding of a denotable is not visible when reducing its definiens to a value. In addition, the decoupled semantics extends this restriction to all of the bindings not lexically visible according to the call-by-need λ_{let}-calculus thereby exposing inherent structure of store. The same can be said about Sestoft's abstract machine and the abstract machines presented in Section 6.2 and 6.3.

8. Extensions

In this section we briefly describe a few common extensions on the λ-calculus. Adding these extensions and applying the equivalence developed here is at the level of an exercise.

8.1 Alias optimization

The introduction of a let expression by Rule (I) corresponds to the dynamic allocation of a delayed application frame. In the case of a denotable, this frame can be eliminated akin to tail-call optimization. We refine Rule (I) as:

$$\begin{array}{r}\langle E[(\lambda x.t)\, x_1],\, X\rangle \to \langle E[t[x_1/x]],\, X\rangle \\ \langle E[(\lambda x.t)\, t_1],\, (x', X)\rangle \to \langle E[\mathsf{let}\ x'= t_1\ \mathsf{in}\ t[x'/x]],\, X\rangle \\ \text{where } t_1 \notin \mathsf{Name}\end{array}$$

This optimization leads to a well-known space optimization of the lazy abstract machine [12, 16, 26].

8.2 Generalized contraction

Often, a contraction of Rule (I) directly gives rise to a new (I) redex. In this case, the series of applications can be done in one step by generalizing Rule (I):

$$(I)^{\mathcal{R}} \begin{array}{l}\langle E[(\lambda x_1.\cdots\lambda x_n.t)\, t_1 \cdots t_n],\, (x'_1,\, (\cdots,\, (x'_n,\, X)))\rangle \\ \to \\ \langle E[\mathsf{let}\ x'_1 = t_1, \cdots, x'_n = t_n\ \mathsf{in}\ t[x'_1/x_1, \cdots, x'_n/x_n]],\, X\rangle\end{array}$$

The resulting abstract machine is a lazy version of Krivine's original abstract machine [31]. (Indeed Krivine's machine implements generalized beta-reduction whereas what is known as the Krivine machine [16] implements ordinary beta-reduction.)

8.3 Preprocessing

Alias optimization and generalized contraction can be further exploited if we split the reduction in two phases: a compile-time notion of reduction $\mathcal{R}_0$:

$$\begin{array}{r}\langle E[t_0\, t_1],\, (x, X)\rangle \to \langle E[\mathsf{let}\ x = t_1\ \mathsf{in}\ t_0\ x],\, X\rangle \\ \text{where } t_1 \notin \mathsf{Name}\end{array}$$

and a run-time notion of reduction $\mathcal{R}_1$ which specializes the rules of $\mathcal{R}$ to the sub-grammar of $\mathcal{R}_0$-normal forms. Indeed, these preprocessed terms are those used by Launchbury [32, Section 3.1] and such preprocessing can be viewed as compiling to a term-graph representation of terms [10, 21].

The global preprocessing of terms invalidates the assumption used to ensure hygiene in Section 6. Since preprocessing occurs under λ-binders, the introduced let-bound names might be duplicated during reduction. Proper hygiene must therefore be ensured by another method, e.g., using explicit substitutions or global renaming.

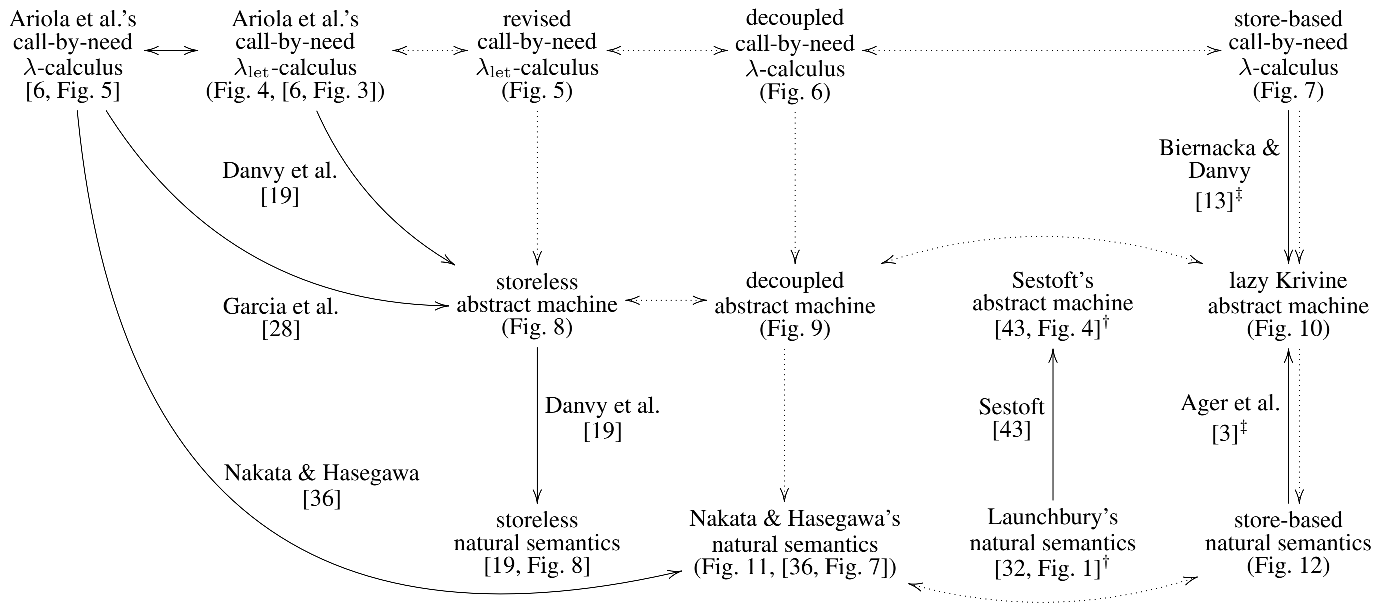
8.4 Cyclic terms

All the inter-derivations of Figure 1 scale to cyclic structures starting with the mutually recursive letrec as defined by Ariola and Felleisen [5]. We are in the process of proving the lock-step equivalences extended for cyclic terms.

9. Conclusion and perspectives

We have presented the first operational account of lazy evaluation that connects theory and practice, where 'theory' stands for a purely syntactic account and 'practice' stands for the traditional implementation technique of imperative memo-thunks. This connection reveals a genuine unity of purpose among theoreticians and implementors and opens the door to more interaction.

Our account is simple, structured and systematic (lock-step equivalence, syntactic correspondence, and functional correspondence). As depicted in Figure 13, it also connects independent forays, discoveries, and inventions. It however does not readily account for issues pertaining to stackability [9, 15], duality [7], and the factorization of an abstract machine into a byte-code compiler and the corresponding virtual machine [1, 24]—a future work.


Ariola et al.'s call-by-need λ-calculus [6, Fig. 5]
Ariola et al.'s call-by-need λ_{let}-calculus (Fig. 4, [6, Fig. 3])
revised call-by-need λ_{let}-calculus (Fig. 5)
decoupled call-by-need λ-calculus (Fig. 6)
store-based call-by-need λ-calculus (Fig. 7)
Danvy et al. [19]
Biernacka & Danvy [13]‡
Garcia et al. [28]
storeless abstract machine (Fig. 8)
decoupled abstract machine (Fig. 9)
Sestoft's abstract machine [43, Fig. 4]†
lazy Krivine abstract machine (Fig. 10)
Danvy et al. [19]
Sestoft [43]
Ager et al. [3]‡
Nakata & Hasegawa [36]
storeless natural semantics [19, Fig. 8]
Nakata & Hasegawa's natural semantics (Fig. 11, [36, Fig. 7])
Launchbury's natural semantics [32, Fig. 1]†
store-based natural semantics (Fig. 12)

Small-step to big-step accounts of call by need are shown from top to bottom.
Storeless to store-based accounts of call by need are shown from left to right.
Full arrows depict existing connections. Dotted arrows depict connections made in the present article.

† For preprocessed terms.

‡ For explicit substitutions / environments

Figure 13: Global picture: semantics for call-by-need evaluation

A. Outline of the correspondences

This appendix briefly summarizes the syntactic correspondence and the functional correspondence used in the body of this article to connect reduction semantics, abstract machines and natural semantics. This summary is based on Danvy's lecture notes at AFP 2008 [17].

A.1 The syntactic correspondence

The syntactic correspondence makes it possible to inter-derive the representation of a reduction semantics and the representation of an abstract machine as pure functional programs.

A reduction semantics is defined with a grammar of terms, a notion of normal form, a collection of potential redexes, a partial contraction function (this function is partial because not all potential redexes are actual ones: terms may be stuck), and a reduction strategy that determines a grammar of reduction contexts. The reduction strategy is implemented with a decomposition function that maps a term in normal form to itself and a term not in normal form to a potential redex and its reduction context. The recomposition function is a left fold over a reduction context. One-step reduction of a term which is not in normal form (1) locates the first potential redex in this term according to the reduction strategy by decomposing this term into a potential redex and its reduction context, (2) contracts this redex if it is an actual one, and (3) recomposes the contractum over the reduction context, yielding a reduct. Evaluation is defined as the iteration of one-step reduction: this iteration enumerates the reduction sequence; it is thus reduction-based. Evaluation can become stuck, or yield a term in normal form, or diverge.

From reduction-based evaluation to reduction-free evaluation: The goal of refocusing is to deforest, rather than enumerate, the reducts of a reduction sequence. To this end, each consecutive call to decompose over the result of a call to recompose is replaced by one call to a refocus function that optimally navigates from a reduction site to the next reduction site. The result is a small-step abstract machine.

From small-step to big-step abstract machine: The small-step abstract machine is transformed into a big-step abstract machine using Ohori and Sasano's lightweight fusion [18, 37].

Transition compression: The corridor transitions of the big-step abstract machine are compressed.

A.2 The functional correspondence

The functional correspondence makes it possible to inter-derive the representation of an evaluation function and the representation of an abstract machine as pure functional programs.

Lambda-lifting: If the evaluation function contains scope-sensitive local functions, their free variables become parameters, and the resulting scope-insensitive functions float up to the top-level lexical scope, yielding recursive equations. Lambda-dropping is the left inverse of lambda-lifting.

Closure conversion: If the recursive equations are higher-order, i.e., use functions as values, these functions are represented as closures, i.e., pairs of terms and lexical environments. Closure unconversion is the left inverse of closure conversion.

Given a compositional evaluation function implementing a denotational semantics, the result of lambda-lifting and closure conversion is a typical functional program implementing a natural semantics.

CPS transformation: All intermediate results are named, their computation is sequentialized (which yields 'A-normal forms'),

$$\frac{}{\square \;\mathbf{C}^{\emptyset}_{\emptyset}\; \square}$$

$$\frac{E \;\mathbf{C}^{\mathbf{X}}_{\mathbf{Y}}\; E'}{E[\square\, t] \;\mathbf{C}^{\mathbf{X}}_{\mathbf{Y}\cup(fv(t)\setminus\mathbf{X})}\; E'[\square\, t]}\text{where } t \in \Lambda$$

$$\frac{E \;\mathbf{C}^{\mathbf{X}}_{\mathbf{Y}}\; E'}{E[\mathsf{let}\ x = t\ \mathsf{in}\ \square] \;\mathbf{C}^{\mathbf{X}\cup\{x\}}_{\mathbf{Y}\cup(fv(t)\setminus\mathbf{X})}\; E'[\mathsf{let}\ x = t\ \mathsf{in}\ \square]}\text{where } t \in \Lambda$$

$$\frac{E_1 \;\mathbf{C}^{\mathbf{X}_1}_{\mathbf{Y}_1}\; E'_1 \qquad E_2 \;\mathbf{C}^{\mathbf{X}_2}_{\mathbf{Y}_2}\; E'_2 \qquad \text{where } x \notin \mathbf{X}_2}{E_1[\mathsf{let}\ x = \square\ \mathsf{in}\ E_2[x]] \;\mathbf{C}^{\mathbf{X}_1}_{\mathbf{Y}_1\cup(\mathbf{Y}_2\setminus\mathbf{X}_1\cup\{x\})}\; E'_1[\mathsf{let}\ x := \square\ \mathsf{in}\ E'_2[x]]}$$

$$\frac{E \;\mathbf{C}^{\mathbf{X}}_{\emptyset}\; E'}{E[t] \;\mathbf{B}\; E'[t]}\text{where } t \in \Lambda \text{ and } fv(t) \subseteq \mathbf{X}$$

Figure 14: A lock-step relation for $\mathcal{R}$ and $\mathcal{C}$

$$\frac{}{\square \;\mathbf{C}^{\emptyset}_{\emptyset}\; \langle \square,\, \square \rangle}$$

$$\frac{E \;\mathbf{C}^{\mathbf{X}}_{\mathbf{Y}}\; \langle E',\, A \rangle}{E[\square\, t] \;\mathbf{C}^{\mathbf{X}}_{\mathbf{Y}\cup(fv(t)\setminus\mathbf{X})}\; \langle E'[\square\, t],\, A \rangle}\text{where } t \in \Lambda$$

$$\frac{E \;\mathbf{C}^{\mathbf{X}}_{\mathbf{Y}}\; \langle E',\, A \rangle}{E[\mathsf{let}\ x = t\ \mathsf{in}\ \square] \;\mathbf{C}^{\mathbf{X}\cup\{x\}}_{\mathbf{Y}\cup(fv(t)\setminus\mathbf{X})}\; \langle E',\, A[\mathsf{let}\ x = t\ \mathsf{in}\ \square] \rangle}\text{where } t \in \Lambda$$

$$\frac{E_1 \;\mathbf{C}^{\mathbf{X}_1}_{\mathbf{Y}_1}\; \langle E'_1,\, A_1 \rangle \qquad E_2 \;\mathbf{C}^{\mathbf{X}_2}_{\mathbf{Y}_2}\; \langle E'_2,\, A_2 \rangle \qquad \text{where } x \notin \mathbf{X}_2}{E_1[\mathsf{let}\ x := \square\ \mathsf{in}\ E_2[x]] \;\mathbf{C}^{\mathbf{X}_1}_{\mathbf{Y}_1\cup(\mathbf{Y}_2\setminus\mathbf{X}_1\cup\{x\})}\; \langle E'_1[E'_2[\mathsf{let}\ x := \square\ \mathsf{in}\ A_2]],\, A_1 \rangle}$$

$$\frac{E \;\mathbf{C}^{\mathbf{X}}_{\emptyset}\; \langle E',\, A \rangle}{E[t] \;\mathbf{B}\; \langle E'[t],\, A \rangle}\text{where } t \in \Lambda \text{ and } fv(t) \subseteq \mathbf{X}$$

Figure 15: A lock-step relation for $\mathcal{C}$ and $\mathcal{D}$

and all functions are passed an extra function representing the rest of the computation: the continuation. The result of the transformation is in the eponymous Continuation-Passing Style. The direct-style transformation is the left inverse of the CPS transformation.

Defunctionalization: The function space of continuations is partitioned into a sum type. Each introduction of a continuation is transformed into an injection into this sum type, and each elimination of a continuation is transformed into a call to a function dispatching over the sum type. Refunctionalization is the left inverse of defunctionalization.

A.3 Synergy

The functional correspondence and the syntactic correspondence synergize because of the concrete coincidence between the data type of evaluation contexts (obtained by defunctionalizing the continuation of the big-step evaluation function) and the data type of reduction contexts (obtained by defunctionalizing the continuation of the small-step reduction function). The abstract connection between reduction order and evaluation order was first pointed out by Plotkin [40].

B. Lock-step relations

The lock-step relations are listed in Figures 14, 15, and 16.

$$\frac{}{\langle \square,\, \square \rangle \;\mathbf{C}^{\emptyset}_{\emptyset}\; \langle \square,\, \varepsilon \rangle}$$

$$\frac{\langle E,\, A \rangle \;\mathbf{C}^{\mathbf{X}}_{\mathbf{Y}}\; \langle E',\, \sigma \rangle}{\langle E[\square\, t],\, A \rangle \;\mathbf{C}^{\mathbf{X}}_{\mathbf{Y}\cup(fv(t)\setminus\mathbf{X})}\; \langle E'[\square\, t],\, \sigma \rangle}\text{where } t \in \Lambda$$

$$\frac{\langle E,\, A \rangle \;\mathbf{C}^{\mathbf{X}}_{\mathbf{Y}}\; \langle E',\, \sigma \rangle}{\langle E',\, A[\mathsf{let}\ x = t\ \mathsf{in}\ \square] \rangle \;\mathbf{C}^{\mathbf{X}\cup\{x\}}_{\mathbf{Y}\cup(fv(t)\setminus\mathbf{X})}\; \langle E',\, \sigma[x = t] \rangle}\begin{array}{l}\text{where } t \in \Lambda \\ \quad \text{and } x \notin \mathrm{dom}(\sigma)\end{array}$$

$$\frac{\langle E_1,\, A_1 \rangle \;\mathbf{C}^{\mathbf{X}_1}_{\mathbf{Y}_1}\; \langle E'_1,\, \sigma_1 \rangle \qquad \langle E_2,\, A_2 \rangle \;\mathbf{C}^{\mathbf{X}_2}_{\mathbf{Y}_2}\; \langle E'_2,\, \sigma_2 \rangle \qquad \begin{array}{l}\text{where } t \in \Lambda \\ \quad \text{and } x \notin \mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma_2) \\ \quad \text{and } \emptyset = \mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2)\end{array}}{\langle E_1[E_2[\mathsf{let}\ x := \square\ \mathsf{in}\ A_2]],\, A_1 \rangle \;\mathbf{C}^{\mathbf{X}_1}_{\mathbf{Y}_1\cup(\mathbf{Y}_2\setminus\mathbf{X}_1\cup\{x\})}\; \langle E'_1[E'_2[x := \square]],\, \sigma_1[\sigma_2[x = t]] \rangle}$$

$$\frac{\langle E,\, A \rangle \;\mathbf{C}^{\mathbf{X}}_{\emptyset}\; \langle E',\, \sigma \rangle}{\langle E[t],\, A \rangle \;\mathbf{B}\; \langle E'[t],\, \sigma \rangle}\text{where } t \in \Lambda \text{ and } fv(t) \subseteq \mathbf{X}$$

Figure 16: A lock-step relation for $\mathcal{D}$ and $\mathcal{S}$

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Technical Report BRICS RS-03-14, Department of Computer Science, Aarhus University, Aarhus, Denmark, Mar. 2003.

[2] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In D. Miller, editor, *Proceedings of the Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, Aug. 2003. ACM Press.

[3] M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the research report BRICS RS-04-3.

[4] Z. M. Ariola, P. Downen, H. Herbelin, and A. Nakata, Keiko Saurin. Classical call-by-need sequent calculi: The unity of semantic artifacts. In T. Schrijvers and P. Thiemann, editors, *Functional and Logic Programming, 11th International Symposium, FLOPS 2012*, number 7294 in Lecture Notes in Computer Science, pages 32–46, Kobe, Japan, May 2012. Springer.

[5] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.

[6] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In P. Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 233–246, San Francisco, California, Jan. 1995. ACM Press.

[7] Z. M. Ariola, H. Herbelin, and A. Saurin. Classical call-by-need and duality. In L. Ong, editor, *Typed Lambda Calculi and Applications, International Conference, TLCA 2011*, number 6690 in Lecture Notes in Computer Science, pages 27–44, Novi Sad, Serbia, June 2011. Springer.

[8] T. Balabonski. A unified approach to fully lazy sharing. In J. Field and M. Hicks, editors, *Proceedings of the Thirty-Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 469–480, Philadelphia, PA, USA, Jan. 2012. ACM Press.

[9] A. Banerjee and D. A. Schmidt. Stackability in the typed call-by-value lambda calculus. *Science of Computer Programming*, 31(1):47–73, 1998.

[10] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In J. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, number 259 in Lecture Notes in Computer Science, pages 141–158, Eindhoven, The Netherlands, June 1987. Springer-Verlag.

[11] M. Beeler, R. W. Gosper, and R. Schroeppel. HAKMEM. AI Memo 239, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, Feb. 1972. `http://home.pipeline.com/~hbaker1/hakmem/`.

[12] M. Biernacka and O. Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):1–30,

2007. Article #6. Extended version available as the research report BRICS RS-06-3.

[13] M. Biernacka and O. Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007. Extended version available as the research report BRICS RS-06-18.

[14] S. Chang and M. Felleisen. The call-by-need lambda calculus, revisited. In H. Seidl, editor, *Programming Languages and Systems, 21st European Symposium on Programming, ESOP 2012*, Lecture Notes in Computer Science, pages 128–147, Tallinn, Estonia, Mar. 2012. Springer.

[15] S. Chang, D. V. Horn, and M. Felleisen. Evaluating call by need on the control stack. In R. Page, Z. Horváth, and V. Zsók, editors, *Trends in Functional Programming, Volume 11*, number 6546 in Lecture Notes in Computer Science, pages 1–15, Norman, Oklahoma, May 2011. Springer.

[16] P. Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3):209–230, 2007. A preliminary version was presented at the 1990 ACM Conference on Lisp and Functional Programming.

[17] O. Danvy. From reduction-based to reduction-free normalization. In P. Koopman, R. Plasmeijer, and D. 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.

[18] O. Danvy and K. 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.

[19] O. Danvy, K. Millikin, J. Munk, and I. Zerny. On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. *Theoretical Computer Science*, 435:21–42, 2012. A preliminary version was presented at the 10th International Symposium on Functional and Logic Programming (FLOPS 2010).

[20] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, Aarhus, Denmark, Nov. 2004. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[21] O. Danvy and I. Zerny. Three syntactic theories for combinatory graph reduction. In M. Alpuente, editor, *Logic Based Program Synthesis and Transformation, 20th International Symposium, LOPSTR 2010, revised selected papers*, number 6564 in Lecture Notes in Computer Science, pages 1–20, Hagenberg, Austria, July 2010. Springer. Invited talk. Extended version to appear in ACM Transactions on Computational Logic.

[22] S. Diehl, P. Hartel, and P. Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000.

[23] R. Douence and P. Fradet. A systematic study of functional language implementations. *ACM Transactions on Programming Languages and Systems*, 20(2):344–387, 1998.

[24] J. Fairbairn and S. Wray. TIM: a simple, lazy abstract machine to execute supercombinators. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 34–45, Portland, Oregon, Sept. 1987. Springer-Verlag.

[25] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

[26] D. P. Friedman, A. Ghuloum, J. G. Siek, and L. Winebarger. Improving the lazy Krivine machine. *Higher-Order and Symbolic Computation*, 20(3):271–293, 2007.

[27] D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *Third International Colloquium on Automata, Languages, and Programming*, pages 257–284. Edinburgh University Press, Edinburgh, Scotland, July 1976.

[28] R. Garcia, A. Lumsdaine, and A. Sabry. Lazy evaluation and delimited control. *Logical Methods in Computer Science*, 6(3:1):1–39, July 2010. A preliminary version was presented at the Thirty-Sixth Annual ACM Symposium on Principles of Programming Languages (POPL 2009).

[29] T. Hardin, L. Maranget, and B. Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.

[30] P. Henderson and J. H. Morris Jr. A lazy evaluator. In S. L. Graham, editor, *Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 95–103. ACM Press, Jan. 1976.

[31] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[32] J. Launchbury. A natural semantics for lazy evaluation. In S. L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, Jan. 1993. ACM Press.

[33] J.-J. Lévy. *Réductions correctes et optimales dans le lambda-calcul*. Thèse d'état, Université de Paris VII, Paris, France, 1978.

[34] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.

[35] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[36] K. Nakata and M. Hasegawa. Small-step and big-step semantics for call-by-need. *Journal of Functional Programming*, 19(6):699–722, 2009.

[37] A. Ohori and I. Sasano. Lightweight fusion by fixed point promotion. In M. Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 42, No. 1, pages 143–154, Nice, France, Jan. 2007. ACM Press.

[38] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[39] M. Pirog and D. Biernacki. A systematic derivation of the STG machine verified in Coq. In J. Gibbons, editor, *Haskell '10: Proceedings of the 2010 ACM SIGPLAN Haskell Symposium*, pages 25–36, Baltimore, Maryland, Sept. 2010. ACM Press.

[40] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[41] J. 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 [42].

[42] J. C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[43] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[44] F. Sieczkowski, M. Biernacka, and D. Biernacki. Automating derivations of abstract machines from reduction semantics: A generic formalization of refocusing in Coq. In J. Hage and M. T. Morazán, editors, *Implementation and Application of Functional Languages, – 22nd International Symposium, IFL 2010*, number 6647 in Lecture Notes in Computer Science, pages 72–88, Alphen aan den Rijn, The Netherlands, Sept. 2010. Springer. Revised Selected Papers.

[45] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, 1979.

[46] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Computing Laboratory, Oxford University, Oxford, UK, 1971.