Small-step and big-step semantics for call-by-need

Keiko Nakata*
Institute of Cybernetics, Tallinn University of Technology
Masahito Hasegawa†
Research Institute for Mathematical Sciences, Kyoto University

Abstract

We present natural semantics for acyclic as well as cyclic call-by-need lambda calculi, which are proved equivalent to the reduction semantics given by Ariola and Felleisen. The natural semantics are big-step and use global heaps, where evaluation is suspended and memorized. The reduction semantics are small-step and evaluation is suspended and memorized locally in let-bindings. Thus two styles of formalization describe the call-by-need strategy from different angles.

The natural semantics for the acyclic calculus is revised from the previous presentation by Maraist et al. and its adequacy is ascribed to its correspondence with the reduction semantics, which has been proved equivalent to call-by-name by Ariola and Felleisen. The natural semantics for the cyclic calculus is inspired by that of Launchbury and Sestoft and we state its adequacy using a denotational semantics in the style of Launchbury; adequacy of the reduction semantics for the cyclic calculus is in turn ascribed to its correspondence with the natural semantics.

1 Introduction

In [7] Launchbury studied a natural semantics for a call-by-need lambda calculus with letrec. He showed the semantics adequate using a denotational semantics. Sestoft later revised Launchbury's semantics [12]. The revised semantics correctly enforces variable hygiene. Moreover the α-renaming strategy of the revised semantics is demonstrated to be suitable in the light of possible implementations with heap-based abstract machines.

In [2] Ariola and Felleisen studied an equational theory for an acyclic (non-recursive) call-by-need lambda calculus. The calculus admits the standardization theorem, which gives rise to a reduction semantics for the calculus. The

This paper is dedicated to the memory of Professor Reiji Nakajima (1947-2008).

*Supported by the Estonian Science Foundation grant no. 6940 and the ERDF cofunded project EXCS, the Estonian Centre of Excellence in Computer Science.

†Partly supported by the Grant-in-Aid for Scientific Research (C) 20500010.

call-by-need evaluator, induced by the theory, is proved equivalent to the call-by-name evaluator of Plotkin [11]; as a result, the reduction semantics is shown to be adequate. Ariola and Felleisen also presented a cyclic (recursive) call-by-need lambda calculus with letrec; however the cyclic calculus has not been explored. For instance, to the best of our knowledge, it has not been known if the calculus relates to call-by-name or if the standard reduction relation, obtained from the one-step reduction relation and evaluation contexts, is adequate.

The two styles of formalization, namely the natural semantics and the reduction semantics, describe the operational semantics for call-by-need from different angles. The natural semantics is big-step and evaluation is suspended and memorized in a global heap. Sestoft's semantics rigorously preserves binding structure, by performing α-renaming when allocating fresh locations in a heap. As he demonstrated by deriving abstract machines from the natural semantics, this approach to variable hygiene has a natural correspondence with possible concrete implementations of call-by-need. The reduction semantics is small-step and evaluation is suspended and memorized locally in let-bindings. It assumes implicit α-conversions. In fact we could think implicit renaming in the reduction semantics is an appropriate approach to variable hygiene, since freshness conditions cannot be checked locally. In other words, the reduction semantics allows for step-wise local reasoning of program behavior using evaluation contexts.

Our work is motivated to bridge the two styles of formalization, both of which we found interesting. Here are contributions of the paper:

- We present natural semantics for acyclic and cyclic call-by-need lambda calculi, and prove them equivalent to the corresponding reduction semantics given by Ariola and Felleisen. For the acyclic calculus we revise the natural semantics given in [9] by correctly enforcing variable hygiene in the style of Sestoft; its adequacy is ascribed to its correspondence with the reduction semantics, which has been proved equivalent to call-by-name by Ariola and Felleisen. The natural semantics for the cyclic calculus is very much inspired by Sestoft's, hence by Launchbury's; the main difference is that our semantics directly works with the full lambda terms with letrec, whereas Sestoft's works with the "normalized" lambda terms, where function arguments are only variables, by having a precompilation step.

- We show the natural semantics for the cyclic calculus adequate by adapting Launchbury's denotational argument. As a consequence the reduction semantics for the cyclic calculus is also shown to be adequate thanks to the equivalence of the two semantics; to the best of our knowledge, this fact has not been shown so far.

In [9] equivalence of the natural semantics and reduction semantics is stated. The paper only mentions that the result is proved by simple induction on derivations in the natural semantics, but we did not find it "simple".

$$
\begin{array}{llll}
\textit{Expressions} & M, N & ::= & x \mid \lambda x.M \mid MN \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N \\
\textit{Values} & V & ::= & \lambda x.M \\
\textit{Answers} & A & ::= & V \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ A \\
\textit{Contexts} & E & ::= & [] \mid EM \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ E \mid \mathsf{let}\ x\ \mathsf{be}\ E\ \mathsf{in}\ E'[x] \\
\textit{Heaps} & \Psi, \Phi & ::= & \epsilon \mid \Psi, x \mapsto M
\end{array}
$$

Figure 1: Syntax of λ_{let}

$$
\begin{array}{ll}
\beta_{need}\text{:} & (\lambda x.M)N \underset{\text{NEED}}{\longrightarrow} \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M \\
\textit{lift}\text{:} & (\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ A)N \underset{\text{NEED}}{\longrightarrow} \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ AN \\
\textit{deref}\text{:} & \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[x] \underset{\text{NEED}}{\longrightarrow} \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[V] \\
\textit{assoc}\text{:} & \mathsf{let}\ x\ \mathsf{be}\ (\mathsf{let}\ y\ \mathsf{be}\ M\ \mathsf{in}\ A)\ \mathsf{in}\ E[x] \underset{\text{NEED}}{\longrightarrow} \mathsf{let}\ y\ \mathsf{be}\ M\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ A\ \mathsf{in}\ E[x]
\end{array}
$$

Figure 2: Reduction semantics for λ_{let}

2 Call-by-need let calculus λ_{let}

We first study the operational semantics for the acyclic (non-recursive) calculus.

2.1 Syntax and Semantics

The syntax of the call-by-need let calculus λ_{let} is defined in figure 1. The reduction and natural semantics are given in figures 2 and 3 respectively. The metavariable X ranges over sets of variables. The notation ϵ denotes an empty sequence. The notation $dom(\Psi)$ denotes the domain of Ψ, namely $dom(\epsilon) = \emptyset$ and $dom(x_1 \mapsto M_1, \ldots, x_n \mapsto M_n) = \{x_1, \ldots, x_n\}$. The notation $M[x'/x]$ denotes substitution of x' for free occurrences of x in M. The notion of free variables is standard and is defined in figure 4. A *program* is a closed expression. We say an expression M *(standard) reduces* to N, written $M \to N$ if $M = E[M']$ and $N = E[N']$ where $M' \underset{\text{NEED}}{\longrightarrow} N'$. We write $M \twoheadrightarrow N$ to denote that M reduces to N in zero or more steps, i.e. $\twoheadrightarrow$ is the reflexive and transitive closure of $\to$.

The reduction semantics is identical to the previous presentation by Ariola and Felleisen [2]. It works with α-equivalence classes of expressions. We assume all binding occurrences of variables in a canonical representative of a class use pairwise distinct names. In particular, evaluation contexts and reduction rules are defined over canonical representatives. Below we recall the reduction semantics briefly. The key rule is β_{need}, where application reduces to a let-construct, thus suspending evaluation of the argument. Since *deref* only substitutes values for variables, β_{need} also ensures that evaluation of an argument is shared among all references to the argument in the function body. The administrative rules *lift* and *assoc* extend the scopes of let-bound variables so that values surrounded by let's become available without duplicating reducible expressions. The following lemma states that there exists at most one partitioning of a program into a context and a redex, namely the unique-decomposition property. It is proved

$$\textit{Lambda}\quad \langle\Psi\rangle\ \lambda x.M \Downarrow_X \langle\Psi\rangle\ \lambda x.M$$

$$\textit{Application}\quad \frac{\langle\Psi\rangle\ M_1 \Downarrow_X \langle\Phi\rangle\ \lambda x.N \quad \langle\Phi, x' \mapsto M_2\rangle\ N[x'/x] \Downarrow_X \langle\Psi'\rangle\ V \quad x' \text{ fresh}}{\langle\Psi\rangle\ M_1 M_2 \Downarrow_X \langle\Psi'\rangle\ V}$$

$$\textit{Let}\quad \frac{\langle\Psi, x' \mapsto N\rangle\ M[x'/x] \Downarrow_X \langle\Phi\rangle\ V \quad x' \text{ fresh}}{\langle\Psi\rangle\ \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M \Downarrow_X \langle\Phi\rangle\ V}$$

$$\textit{Variable}\quad \frac{\langle\Psi\rangle\ M \Downarrow_{X\cup\{x\}\cup dom(\Phi)} \langle\Psi'\rangle\ V}{\langle\Psi, x \mapsto M, \Phi\rangle\ x \Downarrow_X \langle\Psi', x \mapsto V, \Phi\rangle\ V}$$

Figure 3: Natural semantics for λ_{let}

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(\lambda x.M) & = & FV(M)\backslash\{x\} \\ FV(MN) & = & FV(M) \cup FV(N) \\ FV(\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N) & = & FV(M) \cup (FV(N)\backslash\{x\}) \end{array}$$

Figure 4: Free variables

by induction on M.

Lemma 2.1 *For any program M, M is either an answer or there exist a unique context E and a redex N such that $M = E[N]$.*

The natural semantics is revised from that of Maraist et al. [9]. It differs from the previous presentation in the following two points. Firstly our semantics enforces variable hygiene correctly in the style of Sestoft [12] by keeping track of variables which are temporarily deleted from heaps in *Variable* rule. This way, freshness conditions are locally checkable. Secondly our semantics works with the let-explicit calculus instead of the let-free one, hence has an inference rule for the let-construct; this makes it smooth to extend our study of the acyclic calculus to the cyclic calculus in the next section. As in [9] the order of bindings in a heap is significant. That is, re-ordering of bindings in a heap is not allowed. In particular in a heap $x_1 \mapsto M_1, x_2 \mapsto M_2, \ldots, x_n \mapsto M_n$, an expression M_i may contain as free variables only $x_1, \ldots, x_{i-1}$. This explains why it is safe to remove the bindings on the right in *Variable* rule: Φ is not in the scope of M. The natural semantics does not assume implicit α-renaming, but works with (raw) expressions. We may write $\langle\rangle\ M$ to denote $\langle\epsilon\rangle\ M$.

A *configuration* is a pair $\langle\Psi\rangle\ M$ of a heap and an expression. A configuration $\langle x_1 \mapsto M_1, \ldots, x_n \mapsto M_n\rangle\ N$ is closed if $FV(N) \subseteq \{x_1, \ldots, x_n\}$, and $FV(M_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ for any i in $1, \ldots, n$. Borrowing from Sestoft's nomenclature [12],

$$
\begin{array}{l}
\mathsf{let}\ x\ \mathsf{be}\ (\lambda y.y)(\lambda y.y)\ \mathsf{in}\ x \\
\to \mathsf{let}\ x\ \mathsf{be}\ (\mathsf{let}\ y\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ y)\ \mathsf{in}\ x \\
\to \mathsf{let}\ x\ \mathsf{be}\ (\mathsf{let}\ y\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ \lambda y'.y')\ \mathsf{in}\ x \\
\to \mathsf{let}\ y\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ \lambda y'.y'\ \mathsf{in}\ x \\
\to \mathsf{let}\ y\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ \lambda y'.y'\ \mathsf{in}\ \lambda y''.y''
\end{array}
$$

Figure 5: The reduction sequence for $\mathsf{let}\ x\ \mathsf{be}\ (\lambda y.y)(\lambda y.y)\ \mathsf{in}\ x$

$$
\dfrac{\dfrac{\dfrac{\langle\rangle\ \lambda y.y \Downarrow_{\{x'\}} \langle\rangle\ \lambda y.y \quad \dfrac{\langle\rangle\ \lambda y.y \Downarrow_{\{x',y'\}} \langle\rangle\ \lambda y.y}{\langle y' \mapsto \lambda y.y\rangle\ y' \Downarrow_{\{x'\}} \langle y' \mapsto \lambda y.y\rangle\ \lambda y.y}}{\langle\rangle\ (\lambda y.y)(\lambda y.y) \Downarrow_{\{x'\}} \langle y' \mapsto \lambda y.y\rangle\ \lambda y.y}}{\langle x' \mapsto (\lambda y.y)(\lambda y.y)\rangle\ x' \Downarrow_{\emptyset} \langle y' \mapsto \lambda y.y, x' \mapsto \lambda y.y\rangle\ \lambda y.y}}{\langle\rangle\ \mathsf{let}\ x\ \mathsf{be}\ (\lambda y.y)(\lambda y.y)\ \mathsf{in}\ x \Downarrow_{\emptyset} \langle y' \mapsto \lambda y.y, x' \mapsto \lambda y.y\rangle\ \lambda y.y}
$$

Figure 6: The derivation for $\mathsf{let}\ x\ \mathsf{be}\ (\lambda y.y)(\lambda y.y)\ \mathsf{in}\ x$

we say a configuration $\langle x_1 \mapsto M_1, \ldots, x_n \mapsto M_n\rangle\ N$ is *X-good* if $x_1, \ldots, x_n$ are pairwise distinctly named and $\{x_1, \ldots, x_n\}$ and X are disjoint. The judgment $\langle\Psi\rangle\ M \Downarrow_X \langle\Phi\rangle\ V$ is promising if $\langle\Psi\rangle\ M$ is closed and X-good.

Since derivations in the natural semantics only allocate fresh variables in a heap and substitute fresh variables for variables in expressions, a derivation of a promising judgment is promising everywhere. The following lemma is proved by induction on the derivation of $\langle\Psi\rangle\ M \Downarrow_X \langle\Phi\rangle\ V$.

Lemma 2.2 *If $\langle\Psi\rangle\ M$ is closed and X-good and the judgment $\langle\Psi\rangle\ M \Downarrow_X \langle\Phi\rangle\ V$ has a derivation, then $\langle\Phi\rangle\ V$ is closed and X-good, and $dom(\Psi) \subseteq dom(\Phi)$, and every judgment in the derivation is promising.*

Lemma 2.2 shows the natural semantics preserves binding structure in the absence of implicit α-renaming. Since the malloc function returns fresh locations in a heap, the natural semantics indeed relates to heap-based implementations of call-by-need.

Example Figures 5 and 6 present the reduction sequence and the derivation for the expression $\mathsf{let}\ x\ \mathsf{be}\ (\lambda y.y)(\lambda y.y)\ \mathsf{in}\ \mathsf{x}$ respectively.

2.2 Equivalence of the two semantics

The idea underlying our proof is derived from observing the following gap between the two semantics:

- In the reduction semantics heaps are first allocated locally, then are globalized as much as necessary by applying *lift* or *assoc* afterwards to dereference computed values. Besides, the redex is focused implicitly in the sense

$$
\begin{array}{lrcl}
\textit{Frames} & F & ::= & []M \mid \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ [] \mid \mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ E[x] \\
\textit{Structured heaps} & \Sigma & ::= & \epsilon \mid \Sigma, F \\
\textit{Let's} & \Theta & ::= & \epsilon \mid \Theta, \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ []
\end{array}
$$

$$
\frac{}{\vdash \langle\Sigma\rangle\ \lambda x.M \Downarrow \langle\Sigma\rangle\ \lambda x.M}\ \textit{Lam}
$$

$$
\frac{\vdash \langle\Sigma, []M_2\rangle\ M_1 \Downarrow \langle\Sigma_1, []M_2, \Theta\rangle\ \lambda x.N \quad \vdash \langle\Sigma_1, \Theta, \mathsf{let}\ x'\ \mathsf{be}\ M_2\ \mathsf{in}\ []\rangle\ N[x'/x] \Downarrow \langle\Sigma_2\rangle\ V \quad x' \text{ fresh}}{\vdash \langle\Sigma\rangle\ M_1 M_2 \Downarrow \langle\Sigma_2\rangle\ V}\ \textit{App}
$$

$$
\frac{\vdash \langle\Sigma, \mathsf{let}\ x'\ \mathsf{be}\ N\ \mathsf{in}\ []\rangle\ M[x'/x] \Downarrow \langle\Sigma'\rangle\ V \quad x' \text{ fresh}}{\vdash \langle\Sigma\rangle\ \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M \Downarrow \langle\Sigma'\rangle\ V}\ \textit{Letin}
$$

$$
\frac{\vdash \langle\Sigma, \mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ \Sigma_1[x]\rangle\ M \Downarrow \langle\Sigma_2, \mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ \Sigma_1[x], \Theta\rangle\ V}{\vdash \langle\Sigma, \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ [], \Sigma_1\rangle\ x \Downarrow \langle\Sigma_2, \Theta, \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ [], \Sigma_1\rangle\ V}\ \textit{Var}
$$

Figure 7: Instrumented natural semantics for λ_{let}

that the semantics does not specify how to build evaluation contexts, but rather relies on the unique-decomposition property.

- In the natural semantics there is a single global heap. The redex is focused explicitly by applying inference rules, thus decomposing evaluation contexts.

To facilitate reconstructing reduction sequences from derivations by bridging the above gap, our proof introduces an instrumented natural semantics, defined in figure 7, as an intermediary step. The instrumented natural semantics uses *structured heaps* Σ, which are sequences of *frames* F. Intuitively structured heaps are sequenced evaluation contexts.

The notation $LBV(\Sigma)$ denotes the set of variables let-bound in frames of Σ. Or:

$$
\begin{array}{rcl}
LBV(\epsilon) & = & \emptyset \\
LBV(\Sigma, []M) & = & LBV(\Sigma) \\
LBV(\Sigma, \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ []) & = & LBV(\Sigma) \cup \{x\} \\
LBV(\Sigma, \mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ M) & = & LBV(\Sigma)
\end{array}
$$

A structured heap Σ is well-formed if it is an empty sequence, or else $\Sigma = \Sigma', F$ and Σ' is well-formed and one of the following conditions holds:

1. $F = []M$ and $FV(M) \subseteq LBV(\Sigma')$
2. $F = \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ []$ and $FV(M) \subseteq LBV(\Sigma')$ and x is distinct from any of $LBV(\Sigma')$
3. $F = \mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ M$ and $FV(M) \subseteq LBV(\Sigma') \cup \{x\}$ and x is distinct from any of $LBV(\Sigma')$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \langle \mathsf{let}\ x'\ \mathsf{be}\ []\ \mathsf{in}\ x', \mathsf{let}\ y'\ \mathsf{be}\ []\ \mathsf{in}\ y'\rangle\ \lambda y.y \Downarrow \langle \mathsf{let}\ x'\ \mathsf{be}\ []\ \mathsf{in}\ x', \mathsf{let}\ y'\ \mathsf{be}\ []\ \mathsf{in}\ y'\rangle\ \lambda y.y}{\vdash \langle \mathsf{let}\ x'\ \mathsf{be}\ []\ \mathsf{in}\ x', \mathsf{let}\ y'\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ []\rangle\ y' \Downarrow \langle \mathsf{let}\ x'\ \mathsf{be}\ []\ \mathsf{in}\ x', \mathsf{let}\ y'\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ []\rangle\ \lambda y.y}\ (*)}{\vdash \langle \mathsf{let}\ x'\ \mathsf{be}\ []\ \mathsf{in}\ x', []\lambda y.y)\rangle\ \lambda y.y \Downarrow \langle \mathsf{let}\ x'\ \mathsf{be}\ []\ \mathsf{in}\ x', []\lambda y.y)\rangle\ \lambda y.y}}{\vdash \langle \mathsf{let}\ x'\ \mathsf{be}\ []\ \mathsf{in}\ x', \mathsf{let}\ y'\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ []\rangle\ y' \Downarrow \langle \mathsf{let}\ x'\ \mathsf{be}\ []\ \mathsf{in}\ x', \mathsf{let}\ y'\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ []\rangle\ \lambda y.y}\ (*)}$$

$$\dfrac{\dfrac{\dfrac{\vdash \langle \mathsf{let}\ x'\ \mathsf{be}\ []\ \mathsf{in}\ x'\rangle\ (\lambda y.y)(\lambda y.y) \Downarrow \langle \mathsf{let}\ x'\ \mathsf{be}\ []\ \mathsf{in}\ x', \mathsf{let}\ y'\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ []\rangle\ \lambda y.y}{\vdash \langle \mathsf{let}\ x'\ \mathsf{be}\ (\lambda y.y)(\lambda y.y)\ \mathsf{in}\ []\rangle\ x' \Downarrow \langle \mathsf{let}\ y'\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ [], \mathsf{let}\ x'\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ []\rangle\ \lambda y.y}}{\vdash \langle\rangle\ \mathsf{let}\ x\ \mathsf{be}\ (\lambda y.y)(\lambda y.y)\ \mathsf{in}\ x \Downarrow \langle \mathsf{let}\ y'\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ [], \mathsf{let}\ x'\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ []\rangle\ \lambda y.y}$$

Figure 8: The derivation in the instrumented natural semantics for $\mathsf{let}\ x\ \mathsf{be}\ (\lambda y.y)(\lambda y.y)\ \mathsf{in}\ x$

A structured configuration $\langle\Sigma\rangle\ M$ is well-formed if Σ is well-formed and $FV(M) \subseteq LBV(\Sigma)$.

We map structured configurations to expressions by defining translation $\lfloor\cdot\rfloor$ from structured heaps to evaluation contexts:

$$\lfloor\epsilon\rfloor \;=\; [] \qquad \lfloor\Sigma, F\rfloor \;=\; \lfloor\Sigma\rfloor[F]$$

We may identify Σ with $\lfloor\Sigma\rfloor$ when there should be no confusion, thus write $\Sigma[M]$ to denote $\lfloor\Sigma\rfloor[M]$. A (raw) expression $\Sigma[M]$ is not necessarily a canonical representative of an α-equivalence class. The following lemma is proved by induction on the structure of Σ.

Lemma 2.3 *If $\langle\Sigma\rangle\ M$ is well-formed, then $\lfloor\Sigma\rfloor[M]$ is a program.*

Let's look at the inference rules in figure 7. *Lam* and *Letin* are self-explanatory. When evaluating function expression M_1 in *App*, the rule pushes into the heap the frame $[]M_2$, which is popped when evaluating function body N. Notice that the trailing frames to $[]M_2$ in the result heap of the left hypothesis is Θ, which suggests M_1 reduces to an answer $\Theta[\lambda x.N]$. This will be proved in Proposition 2.1. Also, observe the order between Θ and $\mathsf{let}\ x'\ \mathsf{be}\ M_2\ \mathsf{in}\ []$ in the right hypothesis, where let-lifting is performed implicitly. When evaluating variable x in *Var*, the rule pushes the "continuation" $\mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ \Sigma_1[x]$ into the heap. Again, observe the order between Θ and $\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ []$ in the result heap of the consequence, where let-association is implicitly performed. It should be noted that Ariola and Felleisen already observed that Launchbury's formalization has hidden flattening of a heap in his *Variable* rule, which amounts to applying *assoc* [2].

Lemma 2.4 *If $\langle\Sigma\rangle\ M$ is well-formed and $\vdash \langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$, then $\langle\Sigma'\rangle\ V$ is well-formed.*

Proof.

By induction on the derivation of $\vdash \langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$. □

Simple induction proves the instrumented natural semantics correct with respect to the reduction semantics.

Proposition 2.1 *If* $\langle\Sigma\rangle\ M$ *is well-formed and* $\vdash \langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$*, then* $\Sigma[M] \twoheadrightarrow \Sigma'[V]$.

Proof.

By induction on the derivation of $\vdash \langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$ with case analysis on the last rule used.
- The cases of *Lam* and *Letin* are obvious.
- The case of *App*. Suppose we deduce $\vdash \langle\Sigma\rangle\ M_1M_2 \Downarrow \langle\Sigma_2\rangle\ V$ from $\vdash \langle\Sigma, []M_2\rangle\ M_1 \Downarrow \langle\Sigma_1, []M_2, \Theta\rangle\ \lambda x.N$ and $\vdash \langle\Sigma_1, \Theta, \mathsf{let}\ x'\ \mathsf{be}\ M_2\ \mathsf{in}\ []\rangle\ N[x'/x] \Downarrow \langle\Sigma_2\rangle\ V$. Then we have:
$\Sigma[M_1M_2]$
$\twoheadrightarrow \Sigma_1[(\Theta[\lambda x.N])M_2]$ by ind. hyp.
$\twoheadrightarrow \Sigma_1[\Theta[(\lambda x.N)M_2]]$ by *lift*
$\rightarrow \Sigma_1[\Theta[\mathsf{let}\ x'\ \mathsf{be}\ M_2\ \mathsf{in}\ N[x'/x]]]$ by β_{need}
$\twoheadrightarrow \Sigma_2[V]$ by ind. hyp.
- The case of *Var*. Suppose we deduce $\vdash \langle\Sigma, \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ [], \Sigma_1\rangle\ x \Downarrow \langle\Sigma_2, \Theta, \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ [], \Sigma_1\rangle\ V$ from $\vdash \langle\Sigma, \mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ \Sigma_1[x]\rangle\ M \Downarrow \langle\Sigma_2, \mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ \Sigma_1[x], \Theta\rangle\ V$. Then we have:
$\Sigma[\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ \Sigma_1[x]]$
$\twoheadrightarrow \Sigma_2[\mathsf{let}\ x\ \mathsf{be}\ \Theta[V]\ \mathsf{in}\ \Sigma_1[x]]$ by ind. hyp.
$\twoheadrightarrow \Sigma_2[\Theta[\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ \Sigma_1[x]]]$ by *assoc*
$\rightarrow \Sigma_2[\Theta[\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ \Sigma_1[V]]]$ by *deref*

□

We need to prove the original natural semantics in figure 3 correct with respect to the instrumented natural semantics. This is mainly to check that in figure 7 frames are properly pushed and popped so that the pop operation never fails. Below we define a preorder on structured heaps to state that structured heaps only "grow" during derivations.

A preorder $\leq$ on structured heaps is defined such that $F_1, \ldots, F_m \leq F'_1, \ldots, F'_n$ if there is an injection ι from $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$ satisfying the following three conditions:

1. if $i < j$ then $\iota(i) < \iota(j)$
2. for all i in $\{1, \ldots, m\}$, either $F_i = F'_{\iota(i)}$ or else $F_i = \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ []$ and $F'_{\iota(i)} = \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ []$ for some x, M and N
3. for all i in $\{1, \ldots, n\} \backslash ran(\iota)$, $F'_i = \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ []$ for some x and M, where $ran(\iota)$ denotes the range of ι and $\{1, \ldots, n\} \backslash ran(\iota)$ denotes set subtraction.

It is easy to check that $\leq$ is a preorder.

Lemma 2.5 *If* $\langle\Sigma\rangle\ M$ *is well-formed and* $\vdash \langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$*, then* $\Sigma \leq \Sigma'$.

Proof.

By induction on the derivation of $\vdash \langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$. We use the fact that if $\Sigma \leq \Sigma'$ and $\Sigma', \Theta \leq \Sigma''$, then $\Sigma \leq \Sigma''$. □

We define translation $\lceil\cdot\rceil$ from structured heaps to (ordinary) heaps by collecting let-frames as follows:

$$\begin{array}{rcl} \lceil\epsilon\rceil & = & \epsilon \\ \lceil\Sigma, []M\rceil & = & \lceil\Sigma\rceil \\ \lceil\Sigma, \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ []\rceil & = & \lceil\Sigma\rceil, x \mapsto M \\ \lceil\Sigma, \mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ M\rceil & = & \lceil\Sigma\rceil \end{array}$$

Proposition 2.2 *If $\langle\Psi\rangle\ M$ is closed and X-good and $\langle\Psi\rangle\ M \Downarrow_X \langle\Phi\rangle\ V$, then for any Σ such that $\lceil\Sigma\rceil = \Psi$ and $\langle\Sigma\rangle\ M$ is well-formed, $\vdash \langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$ and $\lceil\Sigma'\rceil = \Phi$.*

Proof.

By induction on the derivation of $\langle\Psi\rangle\ M \Downarrow_X \langle\Phi\rangle\ V$ with case analysis on the last rule used.
- The cases of *Lambda* and *Let* are obvious.
- The case of *Application*. Suppose $M = M_1 M_2$ and we deduce $\langle\Psi\rangle\ M_1 M_2 \Downarrow_X \langle\Psi'\rangle\ V$ from $\langle\Psi\rangle\ M_1 \Downarrow_X \langle\Phi\rangle\ \lambda x.N$ and $\langle\Phi, x' \mapsto M_2\rangle\ N[x'/x] \Downarrow_X \langle\Psi'\rangle\ V$. Suppose $\lceil\Sigma\rceil = \Psi$ and $\langle\Sigma\rangle\ M_1 M_2$ is well-formed. By ind. hyp. and Lemma 2.4 and 2.5, $\vdash \langle\Sigma, []M_2\rangle\ M_1 \Downarrow \langle\Sigma_1, []M_2, \Theta\rangle\ \lambda x.N$ and $\lceil\Sigma_1, []M_2, \Theta\rceil = \Phi$ and $\langle\Sigma_1, []M_2, \Theta\rangle\ \lambda x.N$ is well-formed. By ind. hyp.,
$\vdash \langle\Sigma_1, \Theta, \mathsf{let}\ x'\ \mathsf{be}\ M_2\ \mathsf{in}\ []\rangle\ N[x'/x] \Downarrow \langle\Sigma_2\rangle\ V$ and $\lceil\Sigma_2\rceil = \Psi'$.
- The case of *Variable*. Suppose $M = x$ and we deduce $\langle\Psi, x \mapsto N, \Phi\rangle\ x \Downarrow_X \langle\Psi', x \mapsto V, \Phi\rangle\ V$ from $\langle\Psi\rangle\ N \Downarrow_{X \cup \{x\} \cup dom(\Phi)} \langle\Psi'\rangle\ V$. Let $\Sigma = \Sigma_1, \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ [], \Sigma_2$ with $\lceil\Sigma_1\rceil = \Psi$ and $\lceil\Sigma_2\rceil = \Phi$ and $\langle\Sigma\rangle\ x$ well-formed. By ind. hyp. and Lemma 2.5, $\vdash \langle\Sigma_1, \mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ \Sigma_2[x]\rangle\ N \Downarrow \langle\Sigma_3, \mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ \Sigma_2[x], \Theta\rangle\ V$ with $\lceil\Sigma_3, \mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ \Sigma_2[x], \Theta\rceil = \Psi'$. Thus we deduce $\vdash \langle\Sigma\rangle\ x \Downarrow \langle\Sigma_3, \Theta, \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ [], \Sigma_2\rangle\ V$.
□

We prove the reduction semantics correct with respect to the natural semantics without going through the instrumented natural semantics. We first prove three useful lemmas. Lemma 2.6 proves that irrelevant evaluation contexts are replaceable. It lets us prove Lemma 2.7 and 2.8. The former proves that reductions at the function position inside application can be recast outside the application. The latter proves that local reductions inside a let-binding can be recast as top-level reductions. We use the notation $M \twoheadrightarrow^n N$ to denote that M reduces into N in n steps.

Lemma 2.6 *For any Θ, E and x such that $\Theta[E[x]]$ is a program and x is not in $LBV(E)$, if $\Theta[E[x]] \twoheadrightarrow^n \Theta'[E[V]]$, then for any E' such that $\Theta[E'[x]]$ is a program and x is not in $LBV(E')$, $\Theta[E'[x]] \twoheadrightarrow^n \Theta'[E'[V]]$.*

Proof.

By induction on n. Let $\Theta = \Theta_1, \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ [], \Theta_2$ with x not in $LBV(\Theta_2)$. We perform case analysis on the possible reductions of M.
- The case where M is an answer is easy.
- The case where M (one-step) reduces independently of the context is immediate by induction.
- Suppose $M = E_1[x_1]$ and x_1 is not in $LBV(E_1)$ and we have:
$\Theta_1[\mathsf{let}\ x\ \mathsf{be}\ E_1[x_1]\ \mathsf{in}\ \Theta_2[E[x]]] \twoheadrightarrow^{n_1} \Theta_1'[\mathsf{let}\ x\ \mathsf{be}\ E_1[V_1]\ \mathsf{in}\ \Theta_2[E[x]]] \twoheadrightarrow^{n_2} \Theta'[E[V]]$
Then by ind. hyp., we have:
$\Theta_1[\mathsf{let}\ x\ \mathsf{be}\ E_1[x_1]\ \mathsf{in}\ \Theta_2[E'[x]]] \twoheadrightarrow^{n_1} \Theta_1'[\mathsf{let}\ x\ \mathsf{be}\ E_1[V_1]\ \mathsf{in}\ \Theta_2[E'[x]]] \twoheadrightarrow^{n_2} \Theta'[E'[V]]$
□

We introduce a notion of *rooted* reductions to identify a particular intermediate step in reductions: a reduction $M \to M'$ is *β_{need}-rooted with argument N* if $M = \Theta[(\lambda x.N')N]$ and $M' = \Theta[\mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ N']$. A reduction sequence $M \twoheadrightarrow M'$ *preserves a β_{need}-root with argument N* if none of (one-step) reductions in the sequence is β_{need}-rooted with argument N. Intuitively, if $\Theta[MN] \twoheadrightarrow M'$ preserves a β_{need}-root with argument N, then all the reductions only occur at M or in the environment Θ.

Lemma 2.7 *For any Θ, M and N such that $\Theta[MN]$ is a program, if $\Theta[MN] \twoheadrightarrow^n \Theta'[VN]$ and the reduction sequence preserves a β_{need}-root with argument N, then $\Theta[M] \twoheadrightarrow^{n'} \Theta'[V]$ with $n' \leq n$.*

Proof.

By induction on n with case analysis on the possible reductions of M.
- The case where M is an answer is easy.
- The case where M reduces independently of the context is immediate by induction.
- Suppose $M = E[x]$ and x is not in $LBV(E)$ and we have:
$\Theta[(E[x])N] \twoheadrightarrow^{n_1} \Theta_1[(E[V])N] \twoheadrightarrow^{n_2} \Theta'[VN]$
Then by Lemma 2.6 followed by ind. hyp., we have:
$\Theta[E[x]] \twoheadrightarrow^{n_1} \Theta_1[E[V]] \twoheadrightarrow^{n_2'} \Theta'[V]$ where $n_2' \leq n_2$. □

Lemma 2.8 *For any Θ, x, M and E such that $\Theta[\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ E[x]]$ is a program and x is not in $LBV(E)$, if $\Theta[\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ E[x]] \twoheadrightarrow^n \Theta'[\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[x]]$ then $\Theta[M] \twoheadrightarrow^{n'} \Theta'[V]$ with $n' \leq n$.*

Proof.

By induction on n with case analysis on the possible reductions of M.
- The case where M is an answer is easy.
- The case where M reduces independently of the context is immediate by induction.
- Suppose $M = E'[x']$ and x' is not in $LBV(E')$ and we have:
$\Theta[\mathsf{let}\ x\ \mathsf{be}\ E'[x']\ \mathsf{in}\ E[x]] \twoheadrightarrow^{n_1} \Theta_1[\mathsf{let}\ x\ \mathsf{be}\ E'[V']\ \mathsf{in}\ E[x]] \twoheadrightarrow^{n_2} \Theta'[\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[x]]$
Then by Lemma 2.6 followed by ind. hyp., we have:
$\Theta[E'[x']] \twoheadrightarrow^{n_1} \Theta_1[E'[V']] \twoheadrightarrow^{n_2'} \Theta'[V]$ where $n_2' \leq n_2$. □

Now we are ready to prove the reduction semantics correct with respect to the natural semantics, using the above three lemmas to have induction go through.

Proposition 2.3 *For any program M, if $M \twoheadrightarrow A$, then for any X, there exist Θ and V such that $\Theta[V]$ and A belong to the same α-equivalence class and $\langle\rangle\ M \Downarrow_X \langle\lceil\Theta\rceil\rangle\ V$.*

Proof.

Without loss of generality, we assume $\Theta[V]$ and A are syntactically identical. We prove by induction on the length of the reductions of M. Let $M = \Theta'[M']$ with $M' \neq \mathsf{let}\ x\ \mathsf{be}\ N'\ \mathsf{in}\ N$. We perform case analysis on M'.
- The case of abstraction is obvious.
- The case of application. Suppose $M' = M_1 M_2$ and we have:
$\Theta'[M_1 M_2] \twoheadrightarrow \Theta_1[(\lambda x.M_3)M_2] \to \Theta_1[\mathsf{let}\ x\ \mathsf{be}\ M_2\ \mathsf{in}\ M_3] \twoheadrightarrow \Theta[V]$
By Lemma 2.7 and ind. hyp., $\langle\rangle\ \Theta'[M_1] \Downarrow_X \langle\lceil\Theta_1\rceil\rangle\ \lambda x.M_3$. By ind. hyp, $\langle\rangle\ \Theta_1[\mathsf{let}\ x\ \mathsf{be}\ M_2\ \mathsf{in}\ M_3] \Downarrow_X \langle\lceil\Theta\rceil\rangle\ V$. Thus we deduce $\langle\rangle\ \Theta'[M_1 M_2] \Downarrow_X \langle\lceil\Theta\rceil\rangle\ V$.
- The case of a variable. Suppose $M' = x$ and $\Theta' = \Theta_1, \mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ [], \Theta_2$ and we have:
$\Theta_1[\mathsf{let}\ x\ \mathsf{be}\ N\ \mathsf{in}\ \Theta_2[x]] \twoheadrightarrow \Theta_1'[\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ \Theta_2[x]] \to \Theta_1'[\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ \Theta_2[V]]$
By Lemma 2.8 and ind. hyp., $\langle\rangle\ \Theta_1[N] \Downarrow_{X\cup\{x\}\cup dom(\lceil\Theta_2\rceil)} \langle\lceil\Theta_1'\rceil\rangle\ V$, from which we deduce $\langle\rangle\ \Theta'[x] \Downarrow_X \langle\lceil\Theta_1', \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ [], \Theta_2\rceil\rangle\ V$. □

Collecting all propositions together, we prove the equivalence of the two semantics.

Theorem 2.1 *For any program M, the following two conditions hold:*

1. *if $M \twoheadrightarrow A$, then there exist Θ and V such that $\Theta[V]$ and A belong to the same α-equivalence class and $\langle\rangle\ M \Downarrow_\emptyset \langle\lceil\Theta\rceil\rangle\ V$*
2. *if $\langle\rangle\ M \Downarrow_\emptyset \langle\Psi\rangle\ V$, then $M \twoheadrightarrow \Theta[V]$ where $\lceil\Theta\rceil = \Psi$.*

Proof.

1: By Proposition 2.3. 2: By Proposition 2.2 and Lemma 2.5, $\vdash \langle\rangle\ M \Downarrow \langle\Theta\rangle\ V$ with $\lceil\Theta\rceil = \Psi$. By Proposition 2.1, $M \twoheadrightarrow \Theta[V]$. □

3 Call-by-need letrec calculus λ_{letrec}

In this section we extend the equivalence result to the cyclic (recursive) calculus.

Expressions	M, N	::=	$x \mid \lambda x.M \mid MN \mid \mathsf{let\ rec}\ D\ \mathsf{in}\ M \mid \bullet$
Definitions	D	::=	$\epsilon \mid D, x\ \mathsf{be}\ M$
Values	V	::=	$\lambda x.M \mid \bullet$
Answers	A	::=	$V \mid \mathsf{let\ rec}\ D\ \mathsf{in}\ A$
Contexts	E	::=	$[] \mid EM \mid \mathsf{let\ rec}\ D\ \mathsf{in}\ E$
		$\mid$	$\mathsf{let\ rec}\ x\ \mathsf{be}\ E, D\ \mathsf{in}\ E'[x]$
		$\mid$	$\mathsf{let\ rec}\ x'\ \mathsf{be}\ E, D[x, x'], D\ \mathsf{in}\ E'[x]$
Dependencies	$D[x, x']$	::=	$x\ \mathsf{be}\ E[x']$
		$\mid$	$D[x, x''], x''\ \mathsf{be}\ E[x']$

Figure 9: Syntax of λ_{letrec}

3.1 Syntax and semantics

The syntax of the call-by-need letrec calculus λ_{letrec} is defined in figure 9. The reduction and natural semantics are defined in figures 10 and 11 respectively. No ordering among bindings in D is assumed. Metavariables Ψ and Φ range over finite mappings from variables to expressions. Here we do not assume any ordering among bindings in heaps. In particular, a heap may contain cyclic structure such as $\langle x_1 \mapsto \lambda y.x_2 y, x_2 \mapsto \lambda y.x_1 y \rangle$ and $\langle x \mapsto y, y \mapsto x \rangle$. In the natural semantics, the notation $\Psi[x_i \mapsto M_i]_{i \in \{1,\ldots,n\}}$ denotes mapping extension. Precisely,

$$\Psi[x_i \mapsto M_i]_{i \in \{1,\ldots,n\}}(x) = \begin{cases} M_i & \text{when } x = x_i \text{ for some } i \text{ in } 1, \ldots, n \\ \Psi(x) & \text{otherwise} \end{cases}$$

We write $\Psi[x \mapsto M]$ to denote a single extension of Ψ with M at x. In rule *Letrec* of figure 11, M_i''s and N' denote expressions obtained from M_i's and N by substituting x_i''s for x_i's, respectively. We may abbreviate $\langle \Psi \rangle\ M$ where Ψ is an empty mapping, i.e. the domain of Ψ is empty, to $\langle \rangle\ M$. We adapt the definition of free variables in figure 4 for λ_{letrec} by replacing the rule for let with the following rule:

$$\begin{aligned} &FV(\mathsf{let\ rec}\ x_1\ \mathsf{be}\ M_1, \ldots, x_n\ \mathsf{be}\ M_n\ \mathsf{in}\ N) \\ &= (FV(M_1) \cup \ldots \cup FV(M_n) \cup FV(N)) \backslash \{x_1, \ldots, x_n\} \end{aligned}$$

The reduction semantics is mostly identical to the previous presentation by Ariola and Felleisen [2], except that we elaborately deal with "undefinedness", which arises due to direct cycles such as $\mathsf{let\ rec}\ x\ \mathsf{be}\ x\ \mathsf{in}\ M$. Undefinedness represents provable divergences. In our reduction semantics undefinedness, or black holes $\bullet$, are produced and propagated explicitly, in a spirit similar to Wright and Felleisen's treatment of exceptions in a reduction calculus [16]. Rules *error* and $error_{env}$ produce black holes. Applying a black hole to an expression results in a black hole ($error_\beta$). A value may be an abstraction or a black hole. Thus rules *lift*, *deref*, $deref_{env}$, *assoc* and $assoc_{env}$ can be exercised to propagate black holes. Explicit handling of black holes facilitates inductive reasoning. Again the reduction semantics works with α-equivalence classes of expressions. The following lemma states the unique-decomposition property for λ_{letrec} and is proved by induction on M.

$$
\begin{array}{ll}
\beta_{need}: & (\lambda x.M)N \underset{\text{NEED}}{\longrightarrow} \mathsf{let\ rec}\ x\ \mathsf{be}\ N\ \mathsf{in}\ M \\
\mathit{lift}: & (\mathsf{let\ rec}\ D\ \mathsf{in}\ A)N \underset{\text{NEED}}{\longrightarrow} \mathsf{let\ rec}\ D\ \mathsf{in}\ AN \\
\mathit{deref}: & \mathsf{let\ rec}\ x\ \mathsf{be}\ V, D\ \mathsf{in}\ E[x] \underset{\text{NEED}}{\longrightarrow} \mathsf{let\ rec}\ x\ \mathsf{be}\ V, D\ \mathsf{in}\ E[V] \\
\mathit{deref}_{env}: & \mathsf{let\ rec}\ D[x,x'], x'\ \mathsf{be}\ V, D\ \mathsf{in}\ E[x] \underset{\text{NEED}}{\longrightarrow} \mathsf{let\ rec}\ D[x,V], x'\ \mathsf{be}\ V, D\ \mathsf{in}\ E[x] \\
\mathit{assoc}: & \mathsf{let\ rec}\ x\ \mathsf{be}\ (\mathsf{let\ rec}\ D\ \mathsf{in}\ A), D'\ \mathsf{in}\ E[x] \underset{\text{NEED}}{\longrightarrow} \mathsf{let\ rec}\ D, x\ \mathsf{be}\ A, D'\ \mathsf{in}\ E[x] \\
\mathit{assoc}_{env}: & \mathsf{let\ rec}\ x'\ \mathsf{be}\ (\mathsf{let\ rec}\ D\ \mathsf{in}\ A), D[x,x'], D'\ \mathsf{in}\ E[x] \underset{\text{NEED}}{\longrightarrow} \\
& \mathsf{let\ rec}\ D, x'\ \mathsf{be}\ A, D[x,x'], D'\ \mathsf{in}\ E[x] \\
\mathit{error}: & \mathsf{let\ rec}\ D[x,x], D\ \mathsf{in}\ E[x] \underset{\text{NEED}}{\longrightarrow} \mathsf{let\ rec}\ D[x,\bullet], D\ \mathsf{in}\ E[x] \\
\mathit{error}_{env}: & \mathsf{let\ rec}\ D[x',x'], D'[x,x'], D\ \mathsf{in}\ E[x] \underset{\text{NEED}}{\longrightarrow} \mathsf{let\ rec}\ D[x',\bullet], D'[x,x'], D\ \mathsf{in}\ E[x] \\
\mathit{error}_{\beta}: & \bullet M \underset{\text{NEED}}{\longrightarrow} \bullet
\end{array}
$$

Figure 10: Reduction semantics for λ_{letrec}

$$
\frac{}{\langle\Psi\rangle\ V \Downarrow \langle\Psi\rangle\ V}\ \mathit{Value}
$$

$$
\frac{\langle\Psi\rangle\ M_1 \Downarrow \langle\Phi\rangle\ \lambda x.N \quad \langle\Phi[x' \mapsto M_2]\rangle\ N[x'/x] \Downarrow \langle\Psi'\rangle\ V \quad x' \text{ fresh}}{\langle\Psi\rangle\ M_1 M_2 \Downarrow \langle\Psi'\rangle\ V}\ \mathit{Application}
$$

$$
\frac{\langle\Psi[x \mapsto \bullet]\rangle\ \Psi(x) \Downarrow \langle\Phi\rangle\ V}{\langle\Psi\rangle\ x \Downarrow \langle\Phi[x \mapsto V]\rangle\ V}\ \mathit{Variable}
$$

$$
\frac{\langle\Psi[x'_i \mapsto M'_i]_{i\in\{1,\ldots,n\}}\rangle\ N' \Downarrow \langle\Phi\rangle\ V \quad x'_1, \ldots, x'_n \text{ fresh}}{\langle\Psi\rangle\ \mathsf{let\ rec}\ x_1\ \mathsf{be}\ M_1, \ldots, x_n\ \mathsf{be}\ M_n\ \mathsf{in}\ N \Downarrow \langle\Phi\rangle\ V}\ \mathit{Letrec}
$$

$$
\frac{\langle\Psi\rangle\ M_1 \Downarrow \langle\Phi\rangle\ \bullet}{\langle\Psi\rangle\ M_1 M_2 \Downarrow \langle\Phi\rangle\ \bullet}\ \mathit{Error}_{\beta}
$$

Figure 11: Natural semantics for λ_{letrec}

Lemma 3.1 *For any program M, M is either an answer or there exist a unique context E and redex N such that $M = E[N]$.*

The natural semantics is very much inspired by Sestoft's [12], hence by Launchbury's [7]. We revise Sestoft's semantics in the following two points to draw a direct connection with the reduction semantics. Firstly, in accordance with the reduction semantics, our natural semantics may return black holes. In *Variable* rule, x is bound to $\bullet$ while the bound expression to x is evaluated. For instance, $\langle\rangle\ \mathsf{let\ rec}\ x\ \mathsf{be}\ x\ \mathsf{in}\ x \Downarrow \langle x' \mapsto \bullet\rangle\ \bullet$ is deduced in our formulation. Sestoft's formulation removes the binding of x from the heap during its evaluation, thus evaluation involving direct cycles "gets stuck", i.e., no derivation is possible when direct cycles are encountered. Since we do not remove bindings from heaps, freshness conditions are locally checkable without extra variable

$$
\begin{array}{l}
\mathsf{let\ rec}\ x\ \mathsf{be}\ fx, f\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ x \\
\rightarrow \mathsf{let\ rec}\ x\ \mathsf{be}\ (\lambda y.y)x, f\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ x \\
\rightarrow \mathsf{let\ rec}\ x\ \mathsf{be}\ (\mathsf{let\ rec}\ y\ \mathsf{be}\ x\ \mathsf{in}\ y), f\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ x \\
\rightarrow \mathsf{let\ rec}\ x\ \mathsf{be}\ (\mathsf{let\ rec}\ y\ \mathsf{be}\ \bullet\ \mathsf{in}\ y), f\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ x \\
\rightarrow \mathsf{let\ rec}\ x\ \mathsf{be}\ (\mathsf{let\ rec}\ y\ \mathsf{be}\ \bullet\ \mathsf{in}\ \bullet), f\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ x \\
\rightarrow \mathsf{let\ rec}\ y\ \mathsf{be}\ \bullet, x\ \mathsf{be}\ \bullet, f\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ x \\
\rightarrow \mathsf{let\ rec}\ y\ \mathsf{be}\ \bullet, x\ \mathsf{be}\ \bullet, f\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ \bullet
\end{array}
$$

Figure 12: The reduction sequence for $\mathsf{let\ rec}\ x\ \mathsf{be}\ fx, f\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ x$

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x' \mapsto \bullet, f' \mapsto \bullet\rangle\ \lambda y.y \Downarrow \langle x' \mapsto \bullet, f' \mapsto \bullet\rangle\ \lambda y.y}\quad \dfrac{\dfrac{}{\langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle\ \bullet \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle\ \bullet}}{\langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle\ x' \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle\ \bullet}}{\langle x' \mapsto \bullet, f' \mapsto \lambda y.y\rangle\ f' \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y\rangle\ \lambda y.y}\quad \dfrac{\dfrac{}{\langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto x'\rangle\ y' \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle\ \bullet}}{\langle x' \mapsto \bullet, f' \mapsto \lambda y.y\rangle\ f'x' \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle\ \bullet}}{\langle x' \mapsto f'x', f' \mapsto \lambda y.y\rangle\ x' \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle\ \bullet}}{\langle\rangle\ \mathsf{let\ rec}\ x\ \mathsf{be}\ fx, f\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ x \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle\ \bullet}
$$

Figure 13: The derivation for $\mathsf{let\ rec}\ x\ \mathsf{be}\ fx, f\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ x$

tracking. Secondly, we do not precompile expressions into "normalized" ones. Our semantics works with full lambda expressions with letrec, where function arguments may be any expressions, not only variables.

The notation $dom(\Psi)$ denotes the domain of Ψ. A configuration $\langle\Psi\rangle\ M$ is closed if $FV(M) \subseteq dom(\Psi)$, and for any x in $dom(\Psi)$, $FV(\Psi(x)) \subseteq dom(\Psi)$.

Example Figures 12 and 13 present the reduction sequence and the derivation for the expression $\mathsf{let\ rec}\ x\ \mathsf{be}\ fx, f\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ x$ respectively. We deliberately chose a black hole producing expression to demonstrate the difference of our formulation from Ariola and Felleisen's and Sestoft's.

3.2 Equivalence of the two semantics

We prove equivalence of the two semantics for λ_{letrec} in similar steps to those for λ_{let}, and use an instrumented natural semantics defined in figure 14. The notation $\overline{\Theta}$ denotes the flattening of Θ. Or:

$$\overline{\epsilon} \quad = \quad \epsilon \qquad \overline{\Theta, \mathsf{let\ rec}\ D\ \mathsf{in}\ []} \quad = \quad \overline{\Theta}, D$$

The notation $x \in D_{x'}$ denotes that x is letrec-bound in $D_{x'}$, i.e., either x be $[]$ or x be M is in $D_{x'}$. In rule *Letrecin*, M_i''s and N' denote expressions obtained from M_i's and N by substituting x_i''s for x_i's, respectively.

Here a frame may be $\mathsf{let\ rec}\ D\ \mathsf{in}\ []$ or $\mathsf{let\ rec}\ D_x, D\ \mathsf{in}\ E[x]$, instead of $\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ []$ or $\mathsf{let}\ x\ \mathsf{be}\ []\ \mathsf{in}\ E[x]$. We need to adjust the definitions of well-formedness for structured heaps and structured configurations. The notation

$$
\begin{array}{lllll}
\textit{Frames} & F & ::= & []M \mid \mathsf{let\ rec}\ D\ \mathsf{in}\ [] \mid \mathsf{let\ rec}\ D_x, D\ \mathsf{in}\ E[x] \\
\textit{Structured heaps} & \Sigma & ::= & \epsilon \mid \Sigma, F \\
 & D_x & ::= & x\ \mathsf{be}\ [] \mid D[x, x'], x'\ \mathsf{be}\ [] \\
\textit{Letrec's} & \Theta & ::= & \epsilon \mid \Theta, \mathsf{let\ rec}\ D\ \mathsf{in}\ []
\end{array}
$$

$$\textit{Val}\quad \vdash \langle\Sigma\rangle\ V \Downarrow \langle\Sigma\rangle\ V$$

$$\textit{App}\quad \frac{\vdash \langle\Sigma, []M_2\rangle\ M_1 \Downarrow \langle\Sigma_1, []M_2, \Theta\rangle\ \lambda x.N \quad \vdash \langle\Sigma_1, \Theta, \mathsf{let\ rec}\ x'\ \mathsf{be}\ M_2\ \mathsf{in}\ []\rangle\ N[x'/x] \Downarrow \langle\Sigma_2\rangle\ V \quad x'\ \text{fresh}}{\vdash \langle\Sigma\rangle\ M_1 M_2 \Downarrow \langle\Sigma_2\rangle\ V}$$

$$\textit{Letrecin}\quad \frac{\vdash \langle\Sigma, \mathsf{let\ rec}\ x'_1\ \mathsf{be}\ M'_1, \ldots, x'_n\ \mathsf{be}\ M'_n\ \mathsf{in}\ []\rangle\ N' \Downarrow \langle\Sigma'\rangle\ V \quad x'_1, \ldots, x'_n\ \text{fresh}}{\vdash \langle\Sigma\rangle\ \mathsf{let\ rec}\ x_1\ \mathsf{be}\ M_1, \ldots, x_n\ \mathsf{be}\ M_n\ \mathsf{in}\ N \Downarrow \langle\Sigma'\rangle\ V}$$

$$\textit{Var}\quad \frac{\vdash \langle\Sigma, \mathsf{let\ rec}\ x\ \mathsf{be}\ [], D\ \mathsf{in}\ \Sigma_1[x]\rangle\ M \Downarrow \langle\Sigma', \mathsf{let\ rec}\ x\ \mathsf{be}\ [], D'\ \mathsf{in}\ \Sigma_1[x], \Theta\rangle\ V}{\vdash \langle\Sigma, \mathsf{let\ rec}\ x\ \mathsf{be}\ M, D\ \mathsf{in}\ [], \Sigma_1\rangle\ x \Downarrow \langle\Sigma', \mathsf{let\ rec}\ \overline{\Theta}, x\ \mathsf{be}\ V, D'\ \mathsf{in}\ [], \Sigma_1\rangle\ V}$$

$$\textit{Var}_{env}\quad \frac{\vdash \langle\Sigma, \mathsf{let\ rec}\ x\ \mathsf{be}\ [], D_{x'}[\Sigma_1[x]], D\ \mathsf{in}\ E[x']\rangle\ M \Downarrow \langle\Sigma', \mathsf{let\ rec}\ x\ \mathsf{be}\ [], D_{x'}[\Sigma_1[x]], D'\ \mathsf{in}\ E[x'], \Theta\rangle\ V}{\vdash \langle\Sigma, \mathsf{let\ rec}\ x\ \mathsf{be}\ M, D_{x'}, D\ \mathsf{in}\ E[x'], \Sigma_1\rangle\ x \Downarrow \langle\Sigma', \mathsf{let\ rec}\ \overline{\Theta}, x\ \mathsf{be}\ V, D_{x'}, D'\ \mathsf{in}\ E[x'], \Sigma_1\rangle\ V}$$

$$\textit{Err}_{var}\quad \frac{x \in D_{x'}}{\vdash \langle\Sigma, \mathsf{let\ rec}\ D, D_{x'}\ \mathsf{in}\ E[x'], \Sigma'\rangle\ x \Downarrow \langle\Sigma, \mathsf{let\ rec}\ D, D_{x'}\ \mathsf{in}\ E[x'], \Sigma'\rangle\ \bullet}$$

$$\textit{Err}_{\beta}\quad \frac{\vdash \langle\Sigma, []M_2\rangle\ M_1 \Downarrow \langle\Sigma', []M_2, \Theta\rangle\ \bullet}{\vdash \langle\Sigma\rangle\ M_1 M_2 \Downarrow \langle\Sigma', \Theta\rangle\ \bullet}$$

Figure 14: Instrumented natural semantics for λ_{letrec}

$LBV(\Sigma)$ denotes the set of variables letrec-bound in frames of Σ. Or:

$$
\begin{array}{rcl}
LBV(\epsilon) & = & \emptyset \\
LBV(\Sigma, []M) & = & LBV(\Sigma) \\
LBV(\Sigma, \mathsf{let\ rec}\ D\ \mathsf{in}\ []) & = & LBV(\Sigma) \cup LBV(D) \\
LBV(\Sigma, \mathsf{let\ rec}\ D, D_x\ \mathsf{in}\ M) & = & LBV(\Sigma) \cup LBV(D, D_x) \\
LBV(D, x\ \mathsf{be}\ M) & = & LBV(D) \cup \{x\} \\
LBV(D, x\ \mathsf{be}\ []) & = & LBV(D) \cup \{x\}
\end{array}
$$

The notations $Exp(F)$ and $Exp(\Sigma)$ respectively denote the sets of expressions

that F and Σ contain. Or:

$$\begin{array}{rcl} Exp([]M) & = & \{M\} \\ Exp(\mathsf{let\ rec}\ D\ \mathsf{in}\ []) & = & Exp(D) \\ Exp(\mathsf{let\ rec}\ D, D_x\ \mathsf{in}\ M) & = & \{M\} \cup Exp(D, D_x) \\ Exp(\epsilon) & = & \emptyset \\ Exp(D, x\ \mathsf{be}\ M) & = & Exp(D) \cup \{M\} \\ Exp(D, x\ \mathsf{be}\ []) & = & Exp(D) \\ Exp(\Sigma, F) & = & Exp(\Sigma) \cup Exp(F) \end{array}$$

A structured heap Σ is well-formed if it is an empty sequence, or else $\Sigma = \Sigma', F$, and Σ' is well-formed and one of the following conditions hold:

1. $F = []M$ and $FV(M) \subseteq LBV(\Sigma)$
2. $F = \mathsf{let\ rec}\ x_1\ \mathsf{be}\ M_1, \ldots, x_n\ \mathsf{be}\ M_n\ \mathsf{in}\ []$ and $FV(M_i) \subseteq LBV(\Sigma)$ for all i's, and $x_1, \ldots, x_n$ are pairwise distinctly named, and all x_i's are distinct from any of $LBV(\Sigma')$
3. $F = \mathsf{let\ rec}\ x\ \mathsf{be}\ [], x_1\ \mathsf{be}\ M_1, \ldots, x_n\ \mathsf{be}\ M_n\ \mathsf{in}\ N$ and $FV(N) \subseteq LBV(\Sigma)$ and $FV(M_i) \subseteq LBV(\Sigma)$ for all i's, and $x, x_1, \ldots, x_n$ are pairwise distinctly named, and all x_i's and x are distinct from any of $LBV(\Sigma')$,

A structured configuration $\langle\Sigma\rangle\ M$ is well-formed if Σ is well-formed and $FV(M) \subseteq LBV(\Sigma)$.

We use the same definition as in the previous section for the translation $\lfloor\cdot\rfloor$ from structured heaps to contexts:

$$\lfloor\epsilon\rfloor\ =\ [] \qquad \lfloor\Sigma, F\rfloor\ =\ \lfloor\Sigma\rfloor[F]$$

Again we may identify Σ with $\lfloor\Sigma\rfloor$, thus write $\Sigma[M]$ to denote $\lfloor\Sigma\rfloor[M]$. The following lemma is proved by induction on the structure of Σ.

Lemma 3.2 *For any well-formed configuration $\langle\Sigma\rangle\ M$, $\Sigma[M]$ is a program.*

Let's look at the inference rules in figure 14. The first four rules are equivalent to the previous four rules in figure 7. Whereas *Var* corresponds to the production $\mathsf{let\ rec}\ x\ \mathsf{be}\ E, D\ \mathsf{in}\ E'[x]$ of evaluation contexts, Var_{env} does to the production $\mathsf{let\ rec}\ x'\ \mathsf{be}\ E, D[x, x'], D\ \mathsf{in}\ E'[x]$. Err_{var} mediates between the natural and reduction semantics when a black hole is produced. Indeed variables letrec-bound in D_x correspond to variables bound to $\bullet$ in a heap in the natural semantics. The instrumented natural semantics keeps the original expressions bound to the variables to facilitate reconstructing reduction sequences from its derivations. Err_β is almost the same as the original rule $Error_\beta$ in figure 11.

Lemma 3.3 *If $\langle\Sigma\rangle\ M$ is well-formed and $\vdash \langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$, then $\langle\Sigma'\rangle\ V$ is well-formed.*

Proof.

By induction on the derivation of $\langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$. $\square$

Easy induction proves the instrumented natural semantics correct with respect to the reduction semantics.

Proposition 3.1 *If* $\langle\Sigma\rangle\ M$ *is well-formed and* $\vdash \langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$*, then* $\Sigma[M] \twoheadrightarrow \Sigma'[V]$.

Proof.

By induction on the derivation of $\vdash \langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$ with case analysis on the last rule used.
- The case of *Val* is obvious.
- The case of *App*. Suppose we deduce $\vdash \langle\Sigma\rangle\ M_1M_2 \Downarrow \langle\Sigma'\rangle\ V$ from $\vdash \langle\Sigma, []M_2\rangle\ M_1 \Downarrow \langle\Sigma_1, []M_2, \Theta\rangle\ \lambda x.N$ and $\vdash \langle\Sigma_1, \Theta, \mathsf{let\ rec}\ x'\ \mathsf{be}\ M_2\ \mathsf{in}\ []\rangle\ N[x'/x] \Downarrow \langle\Sigma'\rangle\ V$. Then we have:
$\Sigma[M_1M_2]$
$\twoheadrightarrow \Sigma_1[(\Theta[\lambda x.N])M_2]$ by ind. hyp.
$\twoheadrightarrow \Sigma_1[\Theta[(\lambda x.N)M_2]]$ by *lift*
$\rightarrow \Sigma_1[\Theta[\mathsf{let\ rec}\ x'\ \mathsf{be}\ M_2\ \mathsf{in}\ N[x'/x]]]$ by β_{need}
$\twoheadrightarrow \Sigma'[V]$ by ind. hyp.
- The case of *Letrecin* is immediate by induction.
- The case of *Var*. Suppose we deduce $\vdash \langle\Sigma, \mathsf{let\ rec}\ x\ \mathsf{be}\ M, D\ \mathsf{in}\ [], \Sigma_1\rangle\ x \Downarrow \langle\Sigma_2, \mathsf{let\ rec}\ \overline{\Theta}, x\ \mathsf{be}\ V, D'\ \mathsf{in}\ [], \Sigma_1\rangle\ V$ from $\vdash \langle\Sigma, \mathsf{let\ rec}\ x\ \mathsf{be}\ [], D\ \mathsf{in}\ \Sigma_1[x]\rangle\ M \Downarrow \langle\Sigma_2, \mathsf{let\ rec}\ x\ \mathsf{be}\ [], D'\ \mathsf{in}\ \Sigma_1[x], \Theta\rangle\ V$. Then we have:
$\Sigma[\mathsf{let\ rec}\ x\ \mathsf{be}\ M, D\ \mathsf{in}\ \Sigma_1[x]]$
$\twoheadrightarrow \Sigma_2[\mathsf{let\ rec}\ x\ \mathsf{be}\ \Theta[V], D'\ \mathsf{in}\ \Sigma_1[x]]$ by ind. hyp.
$\twoheadrightarrow \Sigma_2[\mathsf{let\ rec}\ \overline{\Theta}, x\ \mathsf{be}\ V, D'\ \mathsf{in}\ \Sigma_1[x]]$ by *assoc*
$\rightarrow \Sigma_2[\mathsf{let\ rec}\ \overline{\Theta}, x\ \mathsf{be}\ V, D'\ \mathsf{in}\ \Sigma_1[V]]$ by *deref*
- The case of Var_{env} is similar to the above *Var* case, where we use $assoc_{env}$ and $deref_{env}$ instead of *assoc* and *deref*, respectively.
- The case of Err_{var} (1). Suppose $x = x'$ and we deduce $\vdash \langle\Sigma, \mathsf{let\ rec}\ D, D_x\ \mathsf{in}\ E[x], \Sigma'\rangle\ x \Downarrow \langle\Sigma, \mathsf{let\ rec}\ D, D_x\ \mathsf{in}\ E[x], \Sigma'\rangle\ \bullet$. The side-condition $x \in D_x$ implies $D_x[\Sigma'[x]] = D[x, x]$. Thus we have $\Sigma[\mathsf{let\ rec}\ D, D_x[\Sigma'[x]]\ \mathsf{in}\ E[x]] \rightarrow \Sigma[\mathsf{let\ rec}\ D, D_x[\Sigma'[\bullet]]\ \mathsf{in}\ E[x]]$ by *error*.
- The case of Err_{var} (2). Suppose $x \neq x'$ and we deduce $\vdash \langle\Sigma, \mathsf{let\ rec}\ D, D_{x'}\ \mathsf{in}\ E[x'], \Sigma'\rangle\ x \Downarrow \langle\Sigma, \mathsf{let\ rec}\ D, D_{x'}\ \mathsf{in}\ E[x'], \Sigma'\rangle\ \bullet$. Then $x \in D_{x'}$ implies $D_{x'}[\Sigma'[x]] = D[x', x], D[x, x]$. Thus we have $\Sigma[\mathsf{let\ rec}\ D, D_{x'}[\Sigma'[x]]\ \mathsf{in}\ E[x']] \rightarrow \Sigma[\mathsf{let\ rec}\ D, D_{x'}[\Sigma'[\bullet]]\ \mathsf{in}\ E[x']]$ by $error_{env}$.
- The case of Err_β is easy and similar to *App*. $\square$

Next we prove the instrumented natural semantics correct with respect to the original natural semantics in figure 11. Again this amounts to check that in the instrumented natural semantics pushing and popping frames into heaps are properly balanced. The proof is similar to the previous one for Proposition 2.2, but we extend the preorder $\leq$ on structured heaps to take account of their cyclic structure.

To define the preorder $\leq$ on structured heaps, we use two auxiliary preorders. The preorder $\leq_{\mathcal{D}}$ on sequences of bindings is defined such that $D \leq_{\mathcal{D}} D'$ if $LBV(D) \subseteq LBV(D')$. The preorder $\leq_{\mathcal{F}}$ on frames is the smallest reflexive and transitive relation satisfying the condition that if $D \leq_{\mathcal{D}} D'$, then $\mathsf{let\ rec}\ D_x, D\ \mathsf{in}\ E[x] \leq_{\mathcal{F}} \mathsf{let\ rec}\ D_x, D'\ \mathsf{in}\ E[x]$ and $\mathsf{let\ rec}\ D\ \mathsf{in}\ [] \leq_{\mathcal{F}} \mathsf{let\ rec}\ D'\ \mathsf{in}\ []$. Then the preorder $\leq$ on structured heaps is defined such that $F_1, \ldots, F_m \leq F'_1, \ldots, F'_n$ if there is an injection ι from $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$ satisfying the following three conditions:

1. if $i < j$ then $\iota(i) < \iota(j)$
2. for all i in $\{1, \ldots, m\}$, $F_i \leq_{\mathcal{F}} F'_{\iota(i)}$
3. for all i in $\{1, \ldots, n\} \backslash ran(\iota)$, $F'_i = \mathsf{let\ rec}\ D\ \mathsf{in}\ []$ for some D.

It is easy to check that $\leq$ is a preorder. The following lemma is proved by induction on the derivation of $\vdash \langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$.

Lemma 3.4 *If $\langle\Sigma\rangle\ M$ is well-formed and $\vdash \langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$, then $\Sigma \leq \Sigma'$.*

We define translation $\lceil\cdot\rceil$ from structured heaps into sequences of bindings by:

$$\begin{array}{rcl} \lceil\epsilon\rceil & = & \epsilon \\ \lceil\Sigma, []M\rceil & = & \lceil\Sigma\rceil \\ \lceil\Sigma, \mathsf{let\ rec}\ D\ \mathsf{in}\ []\rceil & = & \lceil\Sigma\rceil, D \\ \lceil\Sigma, \mathsf{let\ rec}\ D, D_x\ \mathsf{in}\ M\rceil & = & \lceil\Sigma\rceil, D, x_1\ \mathsf{be}\ \bullet, \ldots, x_n\ \mathsf{be}\ \bullet \end{array}$$

where $LBV(D_x) = \{x_1, \ldots, x_n\}$. We identify a sequence of bindings D with a heap Ψ such that $LBV(D) = dom(\Psi)$, and for all x in $dom(\Psi)$, $\Psi(x) = M$ iff D contains $x\ \mathsf{be}\ M$. Thus $\lceil\Sigma\rceil$ denotes a heap.

We prove one basic result about the natural semantics: Lemma 3.5 states that extending heaps with irrelevant bindings does not affect derivations and is proved by routine induction. For mappings Ψ, Φ such that $dom(\Psi)$ and $dom(\Phi)$ are disjoint, the notation $\Psi \cup \Phi$ denotes their union, namely $dom(\Psi \cup \Phi) = dom(\Psi) \cup dom(\Phi)$ and:

$$(\Psi \cup \Phi)(x) = \begin{cases} \Psi(x) & \text{when } x \in dom(\Psi) \\ \Phi(x) & \text{when } x \in dom(\Phi) \end{cases}$$

Lemma 3.5 *For any Ψ, Ψ', Φ and M such that $dom(\Psi')$ and $dom(\Phi)$ are disjoint and $\langle\Psi\rangle\ M$ and $\langle\Psi \cup \Psi'\rangle\ M$ are closed, $\langle\Psi\rangle\ M \Downarrow \langle\Phi\rangle\ V$ iff $\langle\Psi \cup \Psi'\rangle\ M \Downarrow \langle\Phi \cup \Psi'\rangle\ V$ and their derivations are of the same depth.*

Proposition 3.2 *If $\langle\Psi\rangle\ M$ is closed and $\langle\Psi\rangle\ M \Downarrow \langle\Phi\rangle\ V$, then for any Σ such that $\lceil\Sigma\rceil = \Psi$ and $\langle\Sigma\rangle\ M$ is well-formed, $\vdash \langle\Sigma\rangle\ M \Downarrow \langle\Sigma'\rangle\ V$ with $\lceil\Sigma'\rceil = \Phi$.*

Proof.

By induction on the depth of the derivation of $\langle\Psi\rangle\ M \Downarrow \langle\Phi\rangle\ V$ with case analysis on the last rule used.
- The case of *Value* is obvious.
- The case of *Application*. Suppose $\lceil\Sigma\rceil = \Psi$ and $\langle\Sigma\rangle\ M_1M_2$ is well-formed and we deduce $\langle\Psi\rangle\ M_1M_2 \Downarrow \langle\Psi'\rangle\ V$ from $\langle\Psi\rangle\ M_1 \Downarrow \langle\Phi\rangle\ \lambda x.N$ and $\langle\Phi[x' \mapsto M_2]\rangle\ N[x'/x] \Downarrow \langle\Psi'\rangle\ V$. By ind. hyp. and Lemma 3.4, $\vdash \langle\Sigma, []M_2\rangle\ M_1 \Downarrow \langle\Sigma_1, []M_2, \Theta\rangle\ \lambda x.N$. with $\lceil\Sigma_1, []M_2, \Theta\rceil = \Phi$. By Lemma 3.3, $\langle\Sigma_1, []M_2, \Theta\rangle\ \lambda x.N$ is well-formed. By ind. hyp., $\vdash \langle\Sigma_1, \Theta, \mathsf{let\ rec}\ x'\ \mathsf{be}\ M_2\ \mathsf{in}\ []\rangle\ N[x'/x] \Downarrow \langle\Sigma_2\rangle\ V$ with $\lceil\Sigma_2\rceil = \Psi'$.
- The cases of *Error*$_\beta$ and *Letrec* are immediate by induction.
- The case of *Variable*. Suppose we deduce $\langle\Psi\rangle\ x \Downarrow \langle\Phi[x \mapsto V]\rangle\ V$ from $\langle\Psi[x \mapsto \bullet]\rangle\ \Psi(x) \Downarrow \langle\Phi\rangle\ V$. Suppose $\lceil\Sigma\rceil = \Psi$ and $\langle\Sigma\rangle\ x$ is well-formed. There are three possible cases.
- - When $\Psi(x) = \bullet$ and $\Sigma = \Sigma_1, \mathsf{let\ rec}\ D, D_{x'}\ \mathsf{in}\ E[x'], \Sigma_2$ with $x \in D_{x'}$. Then we deduce $\vdash \langle\Sigma\rangle\ x \Downarrow \langle\Sigma\rangle\ \bullet$ by *Err*$_{var}$.
- - When $\Psi(x) = N$ and $\Sigma = \Sigma_1, \mathsf{let\ rec}\ x\ \mathsf{be}\ N, D\ \mathsf{in}\ [], \Sigma_2$. By ind. hyp. and Lemma 3.4 and 3.5, $\vdash \langle\Sigma_1, \mathsf{let\ rec}\ x\ \mathsf{be}\ [], D\ \mathsf{in}\ \Sigma_2[x]\rangle\ N \Downarrow \langle\Sigma_1', \mathsf{let\ rec}\ x\ \mathsf{be}\ [], D'\ \mathsf{in}\ \Sigma_2[x], \Theta\rangle\ V$ and $\lceil\Sigma_1', \mathsf{let\ rec}\ x\ \mathsf{be}\ [], D'\ \mathsf{in}\ \Sigma_2[x], \Theta\rceil$ is the restriction of Φ to $LBV(\Sigma_1', \mathsf{let\ rec}\ x\ \mathsf{be}\ [], D'\ \mathsf{in}\ \Sigma_2[x], \Theta)$. Hence by *Var* we deduce $\vdash \langle\Sigma_1, \mathsf{let\ rec}\ x\ \mathsf{be}\ N, D\ \mathsf{in}\ [], \Sigma\rangle\ x \Downarrow \langle\Sigma_1', \mathsf{let\ rec}\ \overline{\Theta}, x\ \mathsf{be}\ V, D'\ \mathsf{in}\ [], \Sigma_2\rangle\ V$ and $\lceil\Sigma_1', \mathsf{let\ rec}\ \overline{\Theta}, x\ \mathsf{be}\ V, D'\ \mathsf{in}\ [], \Sigma_2\rceil = \Phi[x \mapsto V]$.
- - The case where $\Psi(x) = N$ and $\Sigma = \Sigma_1, \mathsf{let\ rec}\ x\ \mathsf{be}\ N, D, D_{x'}\ \mathsf{in}\ E[x'], \Sigma_2$ is similar to the above case, except that we use *Var*$_{env}$ instead of *Var*. □

We prove the reduction semantics correct with respect to the natural semantics by proving three auxiliary results in Lemma 3.6 and 3.7 and Corollary 3.1, which respectively correspond to Lemma 2.8, 2.7 and 2.6 for the acyclic case.

We say a reduction sequence $M \twoheadrightarrow^n N$ is *autonomous* if either $n = 0$, or else the last step is reduced by rules other than *assoc* or *assoc*$_{env}$. These two rules have particular behaviour in that they flatten nested letrec's on request outside; we will restrict the use of the two rules by requiring a reduction sequence to be autonomous. We write $M \twoheadrightarrow\!\!\!\!\!\cdot^n N$ to denote that M reduces into N in n-steps and the reduction sequence is autonomous. We may omit the suffix n when it is irrelevant.

Lemma 3.6 *The following two conditions hold.*

1. *For any* Θ, x, M, D *and* E *such that* $\Theta[\mathsf{let\ rec}\ x\ \mathsf{be}\ M, D\ \mathsf{in}\ [E[x]]]$ *is a program and* x *is not in* $LBV(E)$, $\Theta[\mathsf{let\ rec}\ x\ \mathsf{be}\ M, D\ \mathsf{in}\ [E[x]]] \twoheadrightarrow\!\!\!\!\!\cdot^n \Theta'[\mathsf{let\ rec}\ x\ \mathsf{be}\ A, D'\ \mathsf{in}\ E[x]]$ *iff* $\Theta[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, D\ \mathsf{in}\ M] \twoheadrightarrow^n \Theta'[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, D'\ \mathsf{in}\ A]$

2. *For any* Θ, $D[x_1, x_m]$, M, D *and* E *such that* $\Theta[\mathsf{let\ rec}\ D[x_1, x_m], x_m\ \mathsf{be}\ M, D\ \mathsf{in}\ E[x_1]]$ *is a program and* x_1 *is not in* $LBV(E)$ *and* $LBV(D[x_1, x_m]) = \{x_1, \ldots, x_{m-1}\}$, $\Theta[\mathsf{let\ rec}\ D[x_1, x_m], x_m\ \mathsf{be}\ M, D\ \mathsf{in}\ E[x_1]] \twoheadrightarrow\!\!\!\!\!\cdot^n \Theta'[\mathsf{let\ rec}\ D[x_1, x_m], x_m\ \mathsf{be}\ A, D'\ \mathsf{in}\ E[x_1]]$ *iff* $\Theta[\mathsf{let\ rec}\ x_1\ \mathsf{be}\ \bullet, \ldots, x_m\ \mathsf{be}\ \bullet, D\ \mathsf{in}\ M] \twoheadrightarrow^n \Theta'[\mathsf{let\ rec}\ x_1\ \mathsf{be}\ \bullet, \ldots, x_m\ \mathsf{be}\ \bullet, D'\ \mathsf{in}\ A]$.

Proof.

First we remark that the autonomy condition uniquely determines n in the if case of both the conditions. We prove by simultaneous induction on the length of the reductions with case analysis on the possible reductions.
- The case where M is an answer is obvious.
- The case where M reduces independently of the context is immediate by induction.
- The case where $M = E'[x']$ and $\Theta = \Theta_1, \mathsf{let\ rec}\ x'\ \mathsf{be}\ N, D_1\ \mathsf{in}\ [], \Theta_2$. We only prove the if case in 1. The other cases are similar. Suppose we have:

$\Theta_1[\mathsf{let\ rec}\ x'\ \mathsf{be}\ N, D_1\ \mathsf{in}\ \Theta_2[\mathsf{let\ rec}\ x\ \mathsf{be}\ E'[x'], D\ \mathsf{in}\ [E[x]]]]$
$\mapsto\!\!\!\!\twoheadrightarrow^{n_1} \Theta_1'[\mathsf{let\ rec}\ x'\ \mathsf{be}\ \Theta_3[V], D_1'\ \mathsf{in}\ \Theta_2[\mathsf{let\ rec}\ x\ \mathsf{be}\ E'[x'], D\ \mathsf{in}\ [E[x]]]]$
$\rightarrow\!\!\!\!\twoheadrightarrow^{n_2} \Theta_1'[\mathsf{let\ rec}\ x'\ \mathsf{be}\ V, \overline{\Theta_3}, D_1'\ \mathsf{in}\ \Theta_2[\mathsf{let\ rec}\ x\ \mathsf{be}\ E'[x'], D\ \mathsf{in}\ [E[x]]]]$
$\rightarrow \Theta_1'[\mathsf{let\ rec}\ x'\ \mathsf{be}\ V, \overline{\Theta_3}, D_1'\ \mathsf{in}\ \Theta_2[\mathsf{let\ rec}\ x\ \mathsf{be}\ E'[V], D\ \mathsf{in}\ [E[x]]]]$
$\mapsto\!\!\!\!\twoheadrightarrow^{n_3} \Theta'[\mathsf{let\ rec}\ x\ \mathsf{be}\ A, D'\ \mathsf{in}\ E[x]]$

By ind. hyp., $\Theta_1[\mathsf{let\ rec}\ x'\ \mathsf{be}\ \bullet, D_1\ \mathsf{in}\ N] \rightarrow\!\!\!\!\twoheadrightarrow^{n_1} \Theta_1'[\mathsf{let\ rec}\ x'\ \mathsf{be}\ \bullet, D_1'\ \mathsf{in}\ \Theta_3[V]]$. Hence we have:

$\Theta_1[\mathsf{let\ rec}\ x'\ \mathsf{be}\ N, D_1\ \mathsf{in}\ \Theta_2[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, D\ \mathsf{in}\ E'[x']]]$
$\mapsto\!\!\!\!\twoheadrightarrow^{n_1} \Theta_1'[\mathsf{let\ rec}\ x'\ \mathsf{be}\ \Theta_3[V], D_1'\ \mathsf{in}\ \Theta_2[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, D\ \mathsf{in}\ E'[x']]]$ by ind. hyp.
$\rightarrow\!\!\!\!\twoheadrightarrow^{n_2} \Theta_1'[\mathsf{let\ rec}\ x'\ \mathsf{be}\ V, \overline{\Theta_3}, D_1'\ \mathsf{in}\ \Theta_2[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, D\ \mathsf{in}\ E'[x']]]$ by *assoc*
$\rightarrow \Theta_1'[\mathsf{let\ rec}\ x'\ \mathsf{be}\ V, \overline{\Theta_3}, D_1'\ \mathsf{in}\ \Theta_2[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, D\ \mathsf{in}\ E'[V]]]$ by *deref*
$\rightarrow\!\!\!\!\twoheadrightarrow^{n_3} \Theta'[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, D'\ \mathsf{in}\ A]$ by ind. hyp.

- The cases where $M = E'[x]$ in 1. and where $M = E'[x_i]$ for some i in $1, \ldots, m$ in 2. are immediate by induction.
- The case where $M = E'[x']$ and x' is in $LBV(D)$ for the if case in 1. Suppose we have:

$\Theta[\mathsf{let\ rec}\ x\ \mathsf{be}\ E'[x'], x'\ \mathsf{be}\ N, D_1\ \mathsf{in}\ E[x]]$
$\mapsto\!\!\!\!\twoheadrightarrow^{n_1} \Theta_1[\mathsf{let\ rec}\ x\ \mathsf{be}\ E'[x'], x'\ \mathsf{be}\ \Theta_2[V], D_1'\ \mathsf{in}\ E[x]]$
$\rightarrow\!\!\!\!\twoheadrightarrow^{n_2} \Theta_1[\mathsf{let\ rec}\ x\ \mathsf{be}\ E'[x'], \overline{\Theta_2}, x'\ \mathsf{be}\ V, D_1'\ \mathsf{in}\ E[x]]$
$\rightarrow \Theta_1[\mathsf{let\ rec}\ x\ \mathsf{be}\ E'[V], \overline{\Theta_2}, x'\ \mathsf{be}\ V, D_1'\ \mathsf{in}\ E[x]]$
$\mapsto\!\!\!\!\twoheadrightarrow^{n_3} \Theta'[\mathsf{let\ rec}\ x\ \mathsf{be}\ A, D'\ \mathsf{in}\ E[x]]$

By ind. hyp., $\Theta[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, x'\ \mathsf{be}\ \bullet, D_1\ \mathsf{in}\ N] \rightarrow\!\!\!\!\twoheadrightarrow^{n_1} \Theta_1[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, x'\ \mathsf{be}\ \bullet, D_1'\ \mathsf{in}\ \Theta_2[V]]$. Hence we have:

$\Theta[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, x'\ \mathsf{be}\ N, D_1\ \mathsf{in}\ E'[x']]$
$\mapsto\!\!\!\!\twoheadrightarrow^{n_1} \Theta_1[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, x'\ \mathsf{be}\ \Theta_2[V], D_1'\ \mathsf{in}\ E'[x']]$ by ind. hyp.
$\rightarrow\!\!\!\!\twoheadrightarrow^{n_2} \Theta_1[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, \overline{\Theta_2}, x'\ \mathsf{be}\ V, D_1'\ \mathsf{in}\ E'[x']]$ by *assoc*
$\rightarrow \Theta_1[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, \overline{\Theta_2}, x'\ \mathsf{be}\ V, D_1'\ \mathsf{in}\ E'[V]]$ by *deref*
$\rightarrow\!\!\!\!\twoheadrightarrow \Theta'[\mathsf{let\ rec}\ D'\ \mathsf{in}\ A]$ by ind. hyp.

- The cases where $M = E'[x']$ and x' is in $LBV(D)$ for the only if case in 1. and the if and only if cases in 2. are similar to the above case. □

Corollary 3.1 *For any Θ, E and x such that $\Theta[E[x]]$ is a program and x is not in $LBV(E)$, if $\Theta[E[x]] \rightarrow\!\!\!\!\twoheadrightarrow^n \Theta'[E[V]]$, then for any E' such that $\Theta[E'[x]]$ is a program and x is not in $LBV(E')$, $\Theta[E'[x]] \rightarrow\!\!\!\!\twoheadrightarrow^n \Theta'[E'[V]]$.*

We adapt the definition of rooted reductions in an obvious way by replacing let with let rec. A reduction $M \rightarrow M'$ is β_{need}-rooted with argument N if $M =$

$\Theta[(\lambda x.N')N]$ and $M' = \Theta[\mathsf{let\ rec}\ x\ \mathsf{be}\ N\ \mathsf{in}\ N']$. A reduction sequence $M \twoheadrightarrow M'$ preserves a β_{need}-root with argument N if none of (one-step) reductions in the sequence is β_{need}-rooted with argument N. The following lemma is proved similarly to Lemma 2.7.

Lemma 3.7 *For any Θ, M and N such that $\Theta[MN]$ is a program, if $\Theta[MN] \twoheadrightarrow^n \Theta'[VN]$ and the reduction sequence preserves a β_{need}-root with argument N, then $\Theta[M] \twoheadrightarrow^{n'} \Theta'[V]$ with $n' \leq n$.*

Now we are ready to prove the reduction semantics correct with respect to the natural semantics.

Proposition 3.3 *For any program M, if $M \twoheadrightarrow A$, then there exist Θ and V such that $\Theta[V]$ and A belong to the same α-equivalence class and $\langle\rangle\ M \Downarrow \langle\lceil\Theta\rceil\rangle\ V$.*

Proof.

Without loss of generality, we assume $\Theta[V]$ and A are syntactically identical. We prove by induction on the length of the reductions of M. Let $M = \Theta'[M']$ with $M' \neq \mathsf{let\ rec}\ D\ \mathsf{in}\ N$. We perform case analysis on M'.
- The case of an answer is obvious.
- Suppose $M = M_1M_2$ and we have:
$\Theta'[M_1M_2] \twoheadrightarrow \Theta_1[(\lambda x.N)M_2] \to \Theta_1[\mathsf{let\ rec}\ x\ \mathsf{be}\ M_2\ \mathsf{in}\ N] \twoheadrightarrow \Theta[V]$

By Lemma 3.7 and ind. hyp., $\langle\rangle\ \Theta'[M_1] \Downarrow \langle\lceil\Theta_1\rceil\rangle\ \lambda x.N$. By ind. hyp., $\langle\rangle\ \Theta_1[\mathsf{let\ rec}\ x\ \mathsf{be}\ M_2\ \mathsf{in}\ N] \Downarrow \langle\lceil\Theta\rceil\rangle\ V$. Thus we deduce $\langle\rangle\ \Theta'[M_1M_2] \Downarrow \langle\lceil\Theta\rceil\rangle\ V$.
- The case where $M = M_1M_2$ and M_1 reduces to $\bullet$ is similar to the above case.
- Suppose $M = x$ and $\Theta = \Theta_1, \mathsf{let\ rec}\ x\ \mathsf{be}\ N, D\ \mathsf{in}\ [], \Theta_2$ and we have:
$\Theta_1[\mathsf{let\ rec}\ x\ \mathsf{be}\ N, D\ \mathsf{in}\ \Theta_2[x]]$
$\twoheadrightarrow^n\ \Theta_1'[\mathsf{let\ rec}\ x\ \mathsf{be}\ \Theta_3[V], D_1\ \mathsf{in}\ \Theta_2[x]]$
$\twoheadrightarrow \Theta_1'[\mathsf{let\ rec}\ x\ \mathsf{be}\ V, \overline{\Theta_3}, D_1\ \mathsf{in}\ \Theta_2[x]]$
$\to \Theta_1'[\mathsf{let\ rec}\ x\ \mathsf{be}\ V, \overline{\Theta_3}, D_1\ \mathsf{in}\ \Theta_2[V]]$

By Lemma 3.6, $\Theta_1[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, D\ \mathsf{in}\ N] \twoheadrightarrow^n\ \Theta_1'[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, D_1\ \mathsf{in}\ \Theta_3[V]]$. By ind. hyp., $\langle\rangle\ \Theta_1[\mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, D\ \mathsf{in}\ N] \Downarrow \langle\lceil\Theta_1', \mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, D_1\ \mathsf{in}\ [], \Theta_3\rceil\rangle\ V$. By Lemma 3.5, $\langle\lceil\Theta_1, \mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, D\ \mathsf{in}\ [], \Theta_2\rceil\rangle\ N \Downarrow \langle\lceil\Theta_1', \mathsf{let\ rec}\ x\ \mathsf{be}\ \bullet, D_1\ \mathsf{in}\ [], \Theta_3, \Theta_2\rceil\rangle\ V$. Thus we deduce $\langle\rangle\ \Theta_1[\mathsf{let\ rec}\ x\ \mathsf{be}\ N, D\ \mathsf{in}\ \Theta_2[x]] \Downarrow \langle\lceil\Theta_1', \mathsf{let\ rec}\ x\ \mathsf{be}\ V, \overline{\Theta_3}, D_1\ \mathsf{in}\ [], \Theta_2\rceil\rangle\ V$.
□

Collecting all propositions together, we prove equivalence of the two semantics.

Theorem 3.1 *For any program M, the following two conditions hold:*

1. *if $M \twoheadrightarrow A$ then there exist Θ and V such that $\Theta[V]$ and A belong to the same α-equivalence class and $\langle\rangle\ M \Downarrow \langle\lceil\Theta\rceil\rangle\ V$*
2. *if $\langle\rangle\ M \Downarrow \langle\Psi\rangle\ V$ then $M \twoheadrightarrow \Theta[V]$ where $\lceil\Theta\rceil = \Psi$.*

Proof.

1: By Proposition 3.3. 2: By Proposition 3.2 and Lemma 3.4, $\vdash \langle\rangle\ M \Downarrow \langle\Theta\rangle\ V$ with $\lceil\Theta\rceil = \Psi$. By Proposition 3.1, $M \twoheadrightarrow \Theta[V]$. □

3.3 Adequacy

In this subsection we state that the natural semantics is adequate using a denotational semantics in the style of Launchbury [7]. We adapt his proof strategy with minor modifications. A gentle explanation of the strategy is referred to his paper.

We define the denotational semantics for *pure* expressions of λ_{letrec}. A program M is *pure* if it does not contain black holes. The denotational semantics models functions by a lifted function space [1]. We represent lifting using Fn, and projection using $\downarrow_{Fn}$ (written as a postfix operator). Let *Values* be some appropriate domain containing at least a lifted version of its own function space. *Environments*, ranged over by ρ, are functions from *Vars* to *Values*, where *Vars* denotes the infinitely many set of variables of λ_{letrec}. The notation $sup(\rho)$ denotes the support of ρ, or $sup(\rho) = \{x \mid \rho(x) \neq \bot\}$. The notation $\{x_1 \mapsto z_1, \ldots, x_n \mapsto z_n\}$ where z_i's are elements of *Values* denotes an environment ρ such that $sup(\rho) = \{x_1, \ldots, x_n\}$ and $\rho(x_i) = z_i$ for all i in $1, \ldots, n$. The notation $\rho_\bot$ denotes an "initial" environment which maps all variables to $\bot$, i.e. $sup(\rho_\bot) = \emptyset$.

The semantic functions $[\![M]\!]_\rho$ and $\{\!\{D\}\!\}_\rho$ respectively give meanings to the expression M and the bindings D under the environment ρ. The former returns an element from *Value* and the latter an environment. They are defined by mutual recursion as follows:

$$
\begin{array}{l}
[\![\lambda x.M]\!]_\rho = Fn\ (\lambda\nu.[\![M]\!]_{\rho \sqcup \{x \mapsto \nu\}}) \\
[\![MN]\!]_\rho = ([\![M]\!]_\rho) \downarrow_{Fn} ([\![N]\!]_\rho) \\
[\![x]\!]_\rho = \rho(x) \\
[\![\mathsf{let\ rec}\ x_1\ \mathsf{be}\ M_1, \ldots, x_n\ \mathsf{be}\ M_n\ \mathsf{in}\ N]\!]_\rho = [\![N]\!]_{\{\!\{x_1\ \mathsf{be}\ M_1, \ldots, x_n\ \mathsf{be}\ M_n\}\!\}_\rho} \\
\{\!\{x_1\ \mathsf{be}\ M_1, \ldots, x_n\ \mathsf{be}\ M_n\}\!\}_\rho = \mu\rho'.\rho \sqcup \{x_1 \mapsto [\![M_1]\!]_{\rho'}, \ldots, x_n \mapsto [\![M_n]\!]_{\rho'}\}
\end{array}
$$

where μ denotes the least fixed point operator. $\{\!\{D\}\!\}_\rho$ is defined only when ρ is consistent with D, i.e., if ρ and D bind the same variable, then they maps the variable to values for which an upper bound exists. The semantic function for heaps is defined in the same way as that for bindings by identifying a heap with an unordered sequence of bindings.

We define an order on environments such that $\rho \leq \rho'$ if for all x in $sup(\rho)$, $\rho(x) = \rho'(x)$.

We revise the natural semantics for λ_{letrec} so that it gets stuck when direct cycles are encountered as in Launchbury's semantics. Therefore we replace the *Variable* rule of figure 11 by the following alternative:

$$
\frac{x \in dom(\Psi) \quad \langle \Psi|_{\overline{x}} \rangle\ \Psi(x) \downarrow_{X \cup \{x\}} \langle \Phi \rangle\ V}{\langle \Psi \rangle\ x \downarrow_X \langle \Phi[x \mapsto V] \rangle\ V}
$$

The notation $\Psi|_{\overline{x}}$ denotes the restriction of Ψ to $dom(\Psi) \backslash \{x\}$. We use $\downarrow$ instead of $\Downarrow$ to denote the revised semantics.

Lemma 3.8 *For any pure expression M, $\langle\rangle\ M \Downarrow \langle \Psi \rangle\ \lambda x.N$ iff $\langle\rangle\ M \downarrow_\emptyset \langle \Psi \rangle\ \lambda x.N$.*

A heap Ψ is pure if for all x in $dom(\Psi)$, $\Psi(x)$ is pure. A configuration $\langle\Psi\rangle\ M$ is pure if both Ψ and M are pure.

Lemma 3.9 *If $\langle\Psi\rangle\ M$ is pure and $\langle\Psi\rangle\ M \downarrow_X \langle\Phi\rangle\ V$, then for any environment ρ, $[\![M]\!]_{\{\Psi\}_\rho} = [\![V]\!]_{\{\Phi\}_\rho}$ and $\{\Psi\}_\rho \leq \{\Phi\}_\rho$.*

The following proposition states that derivations preserve non-bottom meanings of pure expressions.

Proposition 3.4 *For any pure program M, if $\langle\rangle\ M \Downarrow \langle\Psi\rangle\ \lambda x.N$ then $[\![M]\!]_{\rho_\perp} = [\![\lambda x.N]\!]_{\{\Psi\}_{\rho_\perp}}$.*

Proof.

By Lemma 3.8, $\langle\rangle\ M \downarrow_\emptyset \langle\Psi\rangle\ \lambda x.N$. By Lemma 3.9, $[\![M]\!]_{\rho_\perp} = [\![V]\!]_{\{\Psi\}_{\rho_\perp}}$ □

Next we characterize when derivations exist.

Lemma 3.10 *If $\langle\Psi\rangle\ M$ is pure and $\langle\Psi\rangle\ M \downarrow_X \langle\Phi\rangle\ \lambda x.N$ then $[\![M]\!]_{\{\Phi\}_{\rho_\perp}} \neq \perp$.*

Following Launchbury, we define a resourced denotational semantics. Let C be the countable chain domain defined as the least solution to the domain equation $C = C_\perp$. We represent lifting in C by injection function $S : C \to C$ and limit element $S(S(S\ldots))$ by ω. *Resourced environments*, ranged over by σ, are functions from *Vars* to functions from C to *Values*, i.e., $\sigma : Vars \to (C \to Values)$. We define a resourced semantic function $\mathcal{N}[\![M]\!]_\sigma$ as follows:

$$\begin{array}{l}
\mathcal{N}[\![M]\!]_\sigma\ \perp = \perp \\
\mathcal{N}[\![\lambda x.M]\!]_\sigma\ (S\ k) = Fn\ (\lambda\nu.\mathcal{N}[\![M]\!]_{\sigma \sqcup \{x \mapsto \nu\}}) \\
\mathcal{N}[\![MN]\!]_\sigma\ (S\ k) = (\mathcal{N}[\![M]\!]_\sigma\ k) \downarrow_{Fn} (\mathcal{N}[\![N]\!]_\sigma)\ k \\
\mathcal{N}[\![x]\!]_\sigma\ (S\ k) = \sigma\ x\ k \\
\mathcal{N}[\![\mathsf{let\ rec}\ x_1\ \mathsf{be}\ M_1, \ldots, x_n\ \mathsf{be}\ M_n\ \mathsf{in}\ M]\!]_\sigma\ (S\ k) = \\
\mathcal{N}[\![M]\!]_{\mu\sigma'.\sigma \sqcup \{x_1 \mapsto \mathcal{N}[\![M_1]\!]_{\sigma'}, \ldots, x_n \mapsto \mathcal{N}[\![M_n]\!]_{\sigma'}\}}\ k
\end{array}$$

We define an alternative natural semantics in which *Variable* rule is replaced by

$$\frac{\langle\Psi, x \mapsto M\rangle\ M \downarrow_{name} \langle\Phi\rangle\ V}{\langle\Psi, x \mapsto M\rangle\ x \downarrow_{name} \langle\Phi\rangle\ V}$$

We use $\downarrow_{name}$ to denote this alternative semantics.

Lemma 3.11 *For any pure expression M, if $\langle\rangle\ M \downarrow_{name} \langle\Psi\rangle\ \lambda x.N$ then $\langle\rangle\ M \Downarrow \langle\Psi'\rangle\ \lambda x.N$.*

Lemma 3.12 *For any pure expressions $M, M_1, \ldots, M_n$,*
if $\mathcal{N}[\![M]\!]_{\mu\sigma.\{x_1 \mapsto \mathcal{N}[\![M_1]\!]_\sigma, \ldots, x_n \mapsto \mathcal{N}[\![M_n]\!]_\sigma\}}\ (S^m\ \perp) \neq \perp$, then $\langle x_1 \mapsto M_1, \ldots, x_n \mapsto M_n\rangle\ M \downarrow_{name} \langle\Psi\rangle\ \lambda x.N$.

Expressions	M, N	::=	$(M, N) \mid \pi_i(M) \mid \ldots$
Values	V	::=	$(V_1, V_2) \mid \ldots$
Contexts	E	::=	$(E, M) \mid (V, E) \mid \pi_i(E) \mid \ldots$

Figure 15: Extension with pairs

prj :	$\pi_i((V_1, V_2)) \underset{\text{NEED}}{\longrightarrow} V_i$
$lift_{\pi}$:	$\pi_i(\mathsf{let\ rec}\ D\ \mathsf{in}\ A) \underset{\text{NEED}}{\longrightarrow} \mathsf{let\ rec}\ D\ \mathsf{in}\ \pi_i(A)$
$lift_{pair_1}$:	$((\mathsf{let\ rec}\ D\ \mathsf{in}\ A), M) \underset{\text{NEED}}{\longrightarrow} \mathsf{let\ rec}\ D\ \mathsf{in}\ (A, M)$
$lift_{pair_2}$:	$(V, \mathsf{let\ rec}\ D\ \mathsf{in}\ A) \underset{\text{NEED}}{\longrightarrow} \mathsf{let\ rec}\ D\ \mathsf{in}\ (V, A)$

Figure 16: Reduction semantics for pairs

The following proposition states that a pure expression evaluates to an abstraction if and only if its meaning is a non-bottom element. Since the natural semantics is deterministic, we can deduce that if a pure expression evaluates to a black hole then its meaning is a bottom element.

Proposition 3.5 *For any pure program M, $[\![M]\!]_{\rho_\perp} \neq \perp$ iff $\langle\rangle\ M \Downarrow \langle\Psi\rangle\ \lambda x.N$.*

Proof.

If: There exists m such that $\mathcal{N}[\![M]\!]_{\sigma_\perp}\ (S^m\ \perp) \neq \perp$. By Lemma 3.12, $\langle\rangle\ M \downarrow_{name} \langle\Psi\rangle\ \lambda x.N$. By Lemma 3.11, $\langle\rangle\ M \Downarrow \langle\Phi\rangle\ \lambda x.N$. Only if: By Proposition 3.4. □

4 An extension with pairs

In this section we extend the cyclic calculus λ_{letrec} with (eager) pairs. The motivation for the extension is to set up a basic framework to study lazy recursive records. Lazy evaluation is used in some programming languages to evaluate recursive records. Hence we think the extension is worth considering.

To accommodate pairs, we extend the syntax of λ_{letrec} as given in figure 15. Now an expression may be a pair (M, N) or projection $\pi_i(M)$. A value may be a pair of values (V_1, V_2). Evaluation contexts contain three new productions (E, M), (V, E) and $\pi_i(E)$. Pairs are evaluated eagerly from left to right.

Figures 16 and 17 respectively give new rules to be added to the reduction and the evaluation semantics, for evaluating and destructing pairs. The two rules in figure 17 and *prj* in figure 16 should be self-explanatory. Heap reconfiguration is implicit in the evaluation semantics, but is explicit in the reduction semantics. That is, $lift_{\pi}$ is hidden in *Projection*, and $lift_{pair_1}$ and $lift_{pair_2}$ are in *Pair*. The equivalence result of the two semantics straightforwardly carries over to the extension.

Theorem 4.1 *For any program M, the following two conditions hold:*

$$\frac{\langle\Psi\rangle\ M_1 \Downarrow \langle\Psi_1\rangle\ V_1 \quad \langle\Psi_1\rangle\ M_2 \Downarrow \langle\Psi_2\rangle\ V_2}{\langle\Psi\rangle\ (M_1, M_2) \Downarrow \langle\Psi_2\rangle\ (V_1, V_2)}\ \textit{Pair}$$

$$\frac{\langle\Psi\rangle\ M \Downarrow \langle\Phi\rangle\ (V_1, V_2)}{\langle\Psi\rangle\ \pi_i(M) \Downarrow \langle\Phi\rangle\ V_i}\ \textit{Projection}$$

Figure 17: Natural semantics for pairs

Expressions	M, N	::=	$x \mid \lambda x.M \mid MN \mid \mathsf{let\ rec}\ D\ \mathsf{in}\ M \mid \bullet$
Definitions	D	::=	$\epsilon \mid D, x\ \mathsf{be}\ M$
Values	V	::=	$\lambda x.M \mid \bullet$
Answers	A	::=	$V \mid \mathsf{let\ rec}\ D\ \mathsf{in}\ A$
Good Answers	G	::=	$\lambda x.M \mid \mathsf{let\ rec}\ D\ \mathsf{in}\ G$
By-value Contexts	E	::=	$[] \mid EM \mid VE \mid \mathsf{let\ rec}\ D\ \mathsf{in}\ E$
		$\mid$	$\mathsf{let\ rec}\ x = E, D\ \mathsf{in}\ E'[x]$
		$\mid$	$\mathsf{let\ rec}\ x' = E, D[x, x'], D\ \mathsf{in}\ E'[x]$
Dependencies	$D[x, x']$	::=	$x\ \mathsf{be}\ E[x']$
		$\mid$	$D[x, x''], x''\ \mathsf{be}\ E[x']$

Figure 18: Syntax of λ^{val}_{letrec}

1. *if* $M \twoheadrightarrow A$ *then there exist* Θ *and* V *such that* $\Theta[V]$ *and* A *belong to the same* α*-equivalence class and* $\langle\rangle\ M \Downarrow \langle\lceil\Theta\rceil\rangle\ V$
2. *if* $\langle\rangle\ M \Downarrow \langle\Psi\rangle\ V$ *then* $M \twoheadrightarrow \Theta[V]$ *where* $\lceil\Theta\rceil = \Psi$.

5 Call-by-value letrec calculus λ^{val}_{letrec}

The delay and force operators as provided in Scheme [13], or OCaml's equivalent lazy and force [8], can be emulated by $\mathsf{let\ rec}\ x\ \mathsf{be}\ M\ \mathsf{in}\ \lambda x'.x$ for $\mathsf{delay}(M)$ and $M(\lambda x.x)$ for $\mathsf{force}(M)$. It is crucial for this encoding that letrec-bindings are evaluated lazily. However, in the presence of ML's traditional value recursion restriction, which requires the right-hand side of recursive bindings to be syntactic values, lazy letrec's are faithful to ML's letrec's. Note that $\mathsf{delay}(M)$ is considered to be a syntactic value. Therefore we are interested in a call-by-value variant of λ_{letrec}, which can model a call-by-value letrec lambda calculus with delay/force operators. For instance Syme's initialization graphs [14], which underlie the object initialization strategy of F# [15], fit in this variant extended with n-tuples, or records.

In figure 18 we define the syntax of λ^{val}_{letrec}, a call-by-value variant of λ_{letrec}. It differs from λ_{letrec} in that evaluation contexts contain the production VE to

It should be noted that the true beta-value axiom is $(\lambda x.M)V = M[V/x]$, as introduced by Plotkin.

$$
\begin{array}{ll}
\beta_{value}: & (\lambda x.M)(\lambda x'.M') \underset{\text{VALUE}}{\longrightarrow} \mathsf{let\ rec}\ x\ \mathsf{be}\ \lambda x'.M'\ \mathsf{in}\ M \\
\mathit{lift}_{arg}: & V(\mathsf{let\ rec}\ D\ \mathsf{in}\ A) \underset{\text{VALUE}}{\longrightarrow} \mathsf{let\ rec}\ D\ \mathsf{in}\ VA \\
\mathit{error}_{arg}: & (\lambda x.M)\bullet \underset{\text{VALUE}}{\longrightarrow} \bullet
\end{array}
$$

Figure 19: Reduction semantics for λ^{val}_{letrec}

force evaluation of arguments. We have introduced *good answers* to distinguish successful termination, which returns abstraction; we will use good answers to state Proposition 5.1. As for the reduction semantics, we replace β_{need} with β_{value} and add two new rules lift_{arg} and error_{arg} as given in figure 19. Otherwise the reduction rules are unchanged from figure 10. An expression M *by-value reduces* to N, written $M \underset{\text{value}}{\rightarrow} N$, if $M = E[M']$ and $N = E[N']$ where $M' \underset{\text{VALUE}}{\longrightarrow} N'$. We write $\underset{\text{value}}{\twoheadrightarrow}$ to denote the reflexive and transitive closure of $\underset{\text{value}}{\rightarrow}$. To avoid confusion we write $\underset{\text{need}}{\twoheadrightarrow}$, instead of $\twoheadrightarrow$, to denote multi-step reductions in λ_{letrec}.

Proposition 5.1 states that λ_{letrec} is more likely to return good answers than λ^{val}_{letrec}. This is not surprising. We prove the proposition by defining the natural semantics for λ^{val}_{letrec} and by relating λ^{val}_{letrec} and λ_{letrec} in terms of the natural semantics.

Proposition 5.1 *For any program M, if $M \underset{\text{value}}{\twoheadrightarrow} G$ then $M \underset{\text{need}}{\twoheadrightarrow} G'$.*

An expression which returns a black hole in λ^{val}_{letrec} may return abstraction in λ_{letrec}, e.g. $\mathsf{let\ rec}\ x\ \mathsf{be}\ (\lambda y.\lambda y'.y)x\ \mathsf{in}\ x$.

6 Related work

Our work builds on previous work by Launchbury [7], Sestoft [12], Ariola and Felleisen [2] and Maraist et al. [9]. The reduction semantics present in the paper are mostly identical to those of Ariola and Felleisen. As to the natural semantics for λ_{let}, we revised that of Maraist et al. by correctly enforcing variable hygiene in the style of Sestoft and by explicitly introducing an inference rule for the let construct. As to the natural semantics for λ_{letrec}, we revised that of Sestoft by eliminating the precompilation step. Adequacy of the natural semantics for λ_{let} is ascribed to its correspondence with the reduction semantics, which is proved equivalent to call-by-name by Ariola and Felleisen. In turn we showed adequacy of the natural semantics for λ_{letrec} by adapting Launchbury's denotational argument. Adequacy of the reduction semantics for λ_{letrec} is then ascribed to its correspondence with the natural semantics; to the best of our knowledge, this fact has not been shown so far. In the above discussed sense, our work extends those previous work.

There are several lines of work which considers other styles of formalization of call-by-need in the presence or absence of recursion. Below we review some

of them. The reader may be interested in the concluding remarks of [9], where Maraist et al. discuss the reduction semantics in relation to other systems.

Recent work by Garcia et al. [6] proposed an abstract machine for the let-free formulation of the acyclic calculus λ_{let}, which is proved equivalent to the reduction semantics of Ariola and Felleisen [2]. They also presented a simulation of the machine by a call-by-value lambda calculus extended with delimited control operators. While developed independently, their abstract machine, in particular the refined one, and our instrumented natural semantics bear similarities in that both manipulate sequenced evaluation contexts while retaining the structural knowledge of a term that has been discovered. More thorough comparison might suggest a means of simulating the cyclic calculus λ_{letrec} using delimited control. This is one direction for future work.

Sestoft revised the natural semantics of Launchbury by enforcing variable hygiene correctly and changing the α-renaming strategy [12]. He derived an abstract machine for call-by-need from the revised semantics. The machine has a small-step semantics and uses global heaps to implement sharing of evaluation. Starting from a simple machine, he refines it to a more efficient machine in several steps. The machine is proved equivalent to his natural semantics. As discussed earlier, the natural semantics for λ_{letrec} is strongly inspired by his semantics.

Okasaki et al. [10] proposed a transformation of call-by-need λ terms, in the absence of recursion, into continuation-passing style, which is proved equivalent to a call-by-need continuation semantics. Sharing of evaluation is implemented by ML-style references, which resemble global heaps.

Ariola and Klop [4] and Ariola and Blom [3] studied equational theories of cyclic lambda calculi by means of cyclic lambda graphs. The former observed that having non-restricted substitution leads to non-confluence and proposed a restriction on substitution to recover confluence. The latter proposed a relaxed notion of confluence which holds in the presence of non-restricted substitution. In [3] a calculus supporting sharing is considered, but a reduction strategy for the calculus is not studied.

Danvy [5] advocates the use of abstract machines as a "natural meeting ground" of various functional implementations of operational semantics, especially the small-step reduction semantics and big-step natural semantics. In a large perspective, our work presented here can be thought as making an analogous case for a destructive, non-functional setting, in which circularly shared computation contributes significant complexities.

7 Conclusion

We have presented natural semantics for acyclic and cyclic call-by-need lambda calculi, which are proved equivalent to the reduction semantics given by Ariola and Felleisen. We observed differences of the two styles of formalization in the treatment of when to reorganize the heap structure and how to focus redexes. The proof uses instrumented natural semantics as mediatory semantics of the

two, in order to bridge these differences by making heap reorganization and redex focusing explicit.

This work is initially motivated to study lazy evaluation strategies for recursive records in terms of the reduction semantics as well as the natural semantics. Therefore we have considered an extension with eager pairs and a call-by-value variant with lazy letrec.

Acknowledgment

We thank the anonymous referees for their careful reviewing and Matthias Felleisen for his editorial support.

References

[1] S. Abramsky and C.-H. L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105(2):159–267, 1993.

[2] Z. Ariola and M. Felleisen. The Call-by-Need Lambda Calculus. *Journal of Functional Programming*, 7(3), 1997.

[3] Z. M. Ariola and S. Blom. Cyclic Lambda Calculi. In *Proc. Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 77–106. Springer, 1997.

[4] Z. M. Ariola and J. W. Klop. Cyclic lambda graph rewriting. In *Proc. Symposium on Logic in Computer Science*, pages 416–425, 1994.

[5] O. Danvy. Defunctionalized Interpreters for Programming Languages. In *Proc. International Conference on Functional Programming*. ACM Press, 2008.

[6] R. Garcia, A Lumsdaine, and A. Sabry. Lazy Evaluation and Delimited Control. In *Proc. the ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*. ACM Press, 2009.

[7] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. the ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, 1993.

[8] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, release 3.11. Software and documentation available on the Web, `http://caml.inria.fr/`, 2008.

[9] J. Maraist, M. Odersky, and P. Wadler. A Call-by-Need Lambda Calculus. *Journal of Functional Programming*, 8(3), 1998.

[10] C. Okasaki, P. Lee, and D. Tarditi. Call-by-need and Continuation-passing Style. *LISP and Symbolic Computation*, 7, 1994.

[11] G. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

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

[13] M. Sperber, R. K. Dybvig, M. Flatt, and A. V. Straaten. Revised^6 Report on the Algorithmic Language Scheme. Available at `http://www.r6rs.org/`, 2007.

[14] D. Syme. Initializing Mutually Referential Abstract Objects: The Value Recursion Challenge. In *Proc. Workshop on ML*, 2005.

[15] D. Syme and J. Margetson. The F# Programming Language, 2008. Software and documentation available on the Web, `http://research.microsoft.com/en-us/um/people/curtisvv/fsharp_default.aspx`.

[16] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.