The Call-by-Need Lambda Calculus (Unabridged)*

John Maraist Martin Odersky Philip Wadler
Universität Karlsruhe† University of Glasgow‡

October 1994

Abstract

We present a calculus that captures the operational semantics of call-by-need. We demonstrate that the calculus is confluent and standardizable and entails the same observational equivalences as call-by-name lambda calculus.

1 Introduction

Procedure calls come in three styles: call-by-value, call-by-name and call-by-need. The first two of these possess elegant models in the form of corresponding lambda calculi. This paper shows that the third may be equipped with a similar model.

The correspondence between call-by-value lambda calculi and strict functional languages (such as the pure subset of Standard ML) is quite good. The call-by-value mechanism of evaluating an argument in advance is well suited for practical use. The correspondence between call-by-name lambda calculi and lazy functional languages (such as Miranda or Haskell) is not so good. Call-by-name re-evaluates an argument each time it is used, which is prohibitively expensive. So lazy languages are implemented using the call-by-need mechanism proposed by Wadsworth [Wad71], which overwrites an argument with its value the first time it is evaluated, avoiding the need for any subsequent evaluation [Tur79, Joh84, KL89, Pey92]

Call-by-need reduction implements the observational behavior of call-by-name in a way that requires no more substitution steps than call-by-value reduction. It seems to give us something for nothing — the rich equational theory of call-by-name without the overhead incurred by re-evaluating arguments. Yet the resulting gap between the conceptual and the implementation calculi can be dangerous since it might lead to program

*Technical Report #28/94, Fakultät für Informatik, Universität Karlsruhe, Germany, October 1994. Also appears as Technical Report ???, Department of Computing Science, University of Glasgow, Scotland.

†Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, Postfach 6980, 76128 Karlsruhe, Germany. Email, {maraist,odersky}@ira.uka.de.

‡Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. Email, wadler@dcs.glasgow.ac.uk.

transformations that drastically increase the complexity of lazy functional programs. In practice, this is dealt with in an ad hoc manner. One uses the laws of call-by-name lambda calculus to convince oneself that the transformations do not alter the meaning of a program, and one uses informal reasoning to ensure that the transformations do not increase the cost of execution.

However, the reasoning required is more subtle than it may at first appear. For instance, in the term

$$\begin{array}{l} \mathsf{let}\ x = 1 + 2\ \mathsf{in} \\ \quad \mathsf{let}\ f = \lambda y.x + y\ \mathsf{in} \\ \quad\quad f\ y + f\ y \end{array}$$

the variable x appears textually only once, but substituting $1+2$ for x in the body of the let will cause $1 + 2$ to be computed twice rather than once.

Experience shows this can be a significant problem in practice. The Glasgow Haskell Compiler is written in Haskell and self-compiled; it makes extensive use of program transformations. One such transformation inadvertently introduced a loss of sharing, causing the symbol table to be rebuilt *each time* an identifier was looked up. The bug was subtle enough that it was not caught until profiling tools later pinpointed the cause of the slowdown [SP94].

The calculus presented here has the following properties.

- It is confluent. Reductions may be applied to any part of a term, including under a lambda, and regardless of order the same normal form will be reached. This property is valuable for modeling program transformations.

- It possesses a notion of standard reduction: a specified, deterministic reduction sequence that will terminate whenever any reduction sequence terminates. This property is valuable for modeling computation.

- It is observationally equivalent to the call-by-name lambda calculus. Here the notion of observation is taken to be reducibility to weak head normal form, as in the lazy lambda calculus of Abramsky and Ong [Ong88, Abr90]. A corollary is that Abramsky and Ong's models are also sound and adequate for our calculus.

- It can be given a natural semantics, similar to the one proposed for lazy lambda calculus by Launchbury [Lau93]. There is a close correspondence between our natural semantics and our standard reduction scheme.

- It can be formulated with or without the use of a let construct. The reduction rules appear more intuitive if a let construct is used, but an equivalent calculus can be formed without let, using the usual equivalence between $\mathsf{let}\ x = M\ \mathsf{in}\ N$ and $(\lambda x.N)M$.

One drawback of our approach is that it does not yield a good model of recursion, such as that yielded by Launchbury's model. This remains a topic for future work.

Our calculus is the only one we know of with all of these properties. In work done independently of ours, Felleisen and Ariola have recently proposed a similar system, which corresponds to the standard reductions of our calculus [AF94]. Like our standard reduction system, their calculus restricts the set of applicable reductions by means of evaluation contexts. Therefore, fewer program transformations can be expressed as equalities, even though the computational properties of both calculi are equivalent.

Several other methods for modeling call-by-need have been studied. Josephs [Jos89] gives a continuation and store-based denotational semantics of lazy evaluation. Purushotaman and Seaman [PS92] give a structured operational semantics of call-by-name PCF with explicit environments that is then shown to be equivalent to a standard denotational semantics for PCF. Launchbury [Lau93] presents a system with a simpler operational semantics and gives in addition rules for recursive let-bindings that capture call-by-need sharing behavior. The key point about all this work is that it provides an operational model of call-by-need, but does not provide anything like a calculus or a reduction system.

Yoshida [Yos93] presents a weak lambda calculus with explicit environments similar to let constructs, and gives an optimal reduction strategy. Her calculus subsumes several of our reduction rules as structural equivalences. However, due to a different notion of observation, reduction in this calculus is not equivalent to reduction to WHNF. A number of researchers [Fie90, ACCL90, Mar91] have studied reductions that preserve sharing in calculi with explicit substitutions, especially in relation to optimal reduction strategies. Having different aims, the resulting calculi are considerably more complex than those presented here.

The rest of this paper is organized as follows. Section 2 reviews the call-by-name calculus, and Section 3 introduces call-by-need. Section 4 asserts the confluence and standard reduction properties. Section 5 shows that the call-by-need calculus is observationally equivalent to the call-by-name calculus. Section 6 reformulates the calculus to show that extra syntax for let is not required. Section 7 presents a natural semantics of call-by-need, and relates it to the notion of standard reduction. Section 8 discusses extensions, and Section 9 concludes.

2 The Call-by-Name Calculus

Figure 1 reviews the call by name lambda calculus. We define the reduction relation $\xrightarrow[\beta]{}$ to be the compatible closure of β, and $\xrightarrow[\beta]{}\!\!\!\!\twoheadrightarrow$ to be the reflexive, transitive closure of $\xrightarrow[\beta]{}$, omitting subscripts when possible without confusion. We write $M \xrightarrow[\beta_s]{} N$ to mean that we have $M \equiv E[\Delta]$, $N \equiv E[\Delta']$ and $\langle \Delta, \Delta' \rangle \in \beta$, with $\xrightarrow[\beta_s]{}\!\!\!\!\twoheadrightarrow$ as the reflexive, transitive closure of $\xrightarrow[\beta_s]{}$.

Throughout this report we use the following notational conventions. We use fv (M) to denote the free identifiers in a term M. A term is *closed* if $\text{fv}(M) = \emptyset$. We use $M \equiv N$ for syntactic equality of terms (modulo α-renaming) and reserve $M = N$ for convertibility. Following Barendregt [Bar81], we work with equivalence classes of α-renamable terms. To avoid name capture problems in substitutions we assume that the bound and free identifiers of a representative term and all its subterms are always distinct. A context

Syntactic Domains

Variables	x, y, z		
Values	V, W	$::=$	$x \mid \lambda x.M$
Terms	L, M, N	$::=$	$V \mid M\ N$
Evaluation Contexts	E	$::=$	$[] \mid E\ M$

Reduction Rule

$$(\beta) \qquad (\lambda x.M)\ N \quad M[x := N]$$

Figure 1: The call-by-name lambda calculus.

Syntactic Domains

Variables	x, y, z		
Values	V, W	$::=$	$x \mid \lambda x.M$
Terms	L, M, N	$::=$	$V \mid M\ N \mid \mathsf{let}\ x = M\ \mathsf{in}\ N$

Reduction Rules

(let-I)	$(\lambda x.M)\ N$	$\to$	$\mathsf{let}\ x = N\ \mathsf{in}\ M$
(let-V)	$\mathsf{let}\ x = V\ \mathsf{in}\ C[x]$	$\to$	$\mathsf{let}\ x = V\ \mathsf{in}\ C[V]$
(let-C)	$(\mathsf{let}\ x = L\ \mathsf{in}\ M)\ N$	$\to$	$\mathsf{let}\ x = L\ \mathsf{in}\ M\ N$
(let-A)	$\mathsf{let}\ y = (\mathsf{let}\ x = L\ \mathsf{in}\ M)\ \mathsf{in}\ N$	$\to$	$\mathsf{let}\ x = L\ \mathsf{in}\ \mathsf{let}\ y = M\ \mathsf{in}\ N$
(let-GC)	$\mathsf{let}\ x = M\ \mathsf{in}\ N$	$\to$	$N \qquad \mathsf{if}\ x \notin fv\ N$

Figure 2: The call-by-need λ-calculus λ_{let}.

$C[\ ]$ is a term with a single hole $[\ ]$ in it. $C[M]$ denotes the term that results from replacing the hole in $C[\ ]$ with M. If R is a notion of reduction, we use $\xrightarrow[R]{}$ to express that M reduces in one R-reduction step to N, and $M \twoheadrightarrow_R N$ to express that M reduces in zero or more R-steps to N. The subscript is omitted if clear from the context.

3 The Call-By-Need Lambda Calculus

Figure 2 details the call-by-need[1] calculus, λ_{let}. We augment the term syntax of λ-calculus with a let-construct. The underlying idea is to represent a reference to a node in a function graph by a let-bound indentifier. Hence, sharing in a function graph corresponds to naming in a term.

The second half of Figure 2 presents reduction rules for λ_{let}.

- Rule let-I introduces a let binding from an application. Given an application $(\lambda x.M)\ N$,

[1] "call-by-need" rather than "lazy" to avoid a name clash with [Abr90] which describes call-by-name reduction to WHNF.

a reducer should construct a copy of the body M where all occurrences of x are replaced by a reference to a single occurrence of the graph of N. let-I models this by representing the reference with a let-bound name.

- Dereferencing is expressed by rule let-V, which substitutes a defining value for a variable occurrence. Note that, since only values are copied, there is no risk of duplicating work in the form of reductions that should have been made to a single, shared expression.

- Rule let-C allows let-bindings to commute with applications, and thus pulls a let-binding out of the function part of an application.

- Rule let-A transforms left-nested let's into right-nested let's. It is a directed version of the associativity law for the call-by-name monad [Mog91].

- Finally, the "garbage collection" rule let-GC drops a let-binding whose defined variable no longer appears in the term. let-GC is not strictly needed for evaluation (as seen in Section 4 where we discuss standard reduction), but it helps to keep terms shorter.

Clearly, these rules never duplicate a term which is not a value. Furthermore, we will show in Section 5 that a term evaluates to an answer in our calculus if and only if it evaluates to an answer in the call-by-name λ-calculus. So λ_{let} fulfills the expectations for what a call-by-need reduction scheme should provide: no loss of sharing except inside values and observational equivalence to the classical call-by-name calculus.

Note also that λ_{let} is an extension of the call-by-value λ-calculus. A βV-reduction

$$(\lambda x.M)\ V \to [V/x]M$$

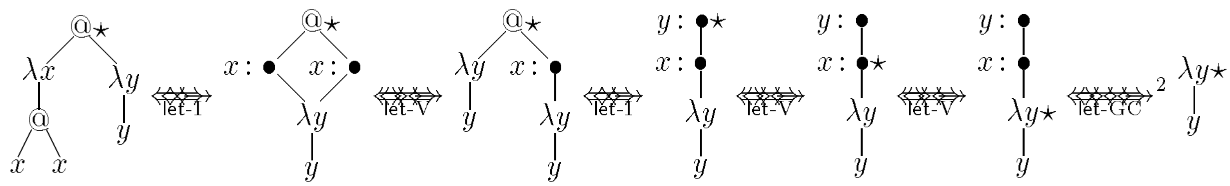
can be expressed by the following sequence of λ_{let}-reductions, where there is one let-V step for each occurrence of x in M.

$$\begin{array}{rll} (\lambda x.M)\ V & \xrightarrow[\text{let-I}]{} & \mathsf{let}\ x = V\ \mathsf{in}\ M \\ & \twoheadrightarrow_{\text{let-V}} & \mathsf{let}\ x = V\ \mathsf{in}\ [V/x]\ M \\ & \xrightarrow[\text{let-GC}]{} & [V/x]\ M \end{array}$$

Example 3.1. $(\lambda x.x\ x)\,(\lambda y.y)$.

$$\begin{array}{llllll} (\lambda x.x\ x)\,(\lambda y.y) & \xrightarrow[\text{let-I}]{} & \begin{array}{l}\mathsf{let}\ x = \lambda y.y \\ \ \mathsf{in}\ x\ x\end{array} & \xrightarrow[\text{let-V}]{} & \begin{array}{l}\mathsf{let}\ x = \lambda y.y \\ \ \mathsf{in}\ (\lambda y.y)\ x\end{array} & \xrightarrow[\text{let-I}]{} \begin{array}{l}\mathsf{let}\ x = \lambda y.y \\ \ \mathsf{in}\ \mathsf{let}\ y = x \\ \quad \mathsf{in}\ y\end{array} \\ & \xrightarrow[\text{let-V}]{} & \begin{array}{l}\mathsf{let}\ x = \lambda y.y \\ \ \mathsf{in}\ \mathsf{let}\ y = x \\ \quad \mathsf{in}\ x\end{array} & \xrightarrow[\text{let-V}]{} & \begin{array}{l}\mathsf{let}\ x = \lambda y.y \\ \ \mathsf{in}\ \mathsf{let}\ y = x \\ \quad \mathsf{in}\ \lambda y.y\end{array} & \xrightarrow[\text{let-GC}]{}^{2}\ \lambda y.y \end{array}$$

Graphically, we have the following sequence, where we mark the node currently considered the root of the graph with a star ($\star$).



4 Syntactic Properties

Lambda calculi have a number of properties that are useful in modeling programming languages, as has been demonstrated by their great success in modeling Algol, Iswim, and a host of successor languages. The confluence property set forth in the Church-Rosser theorem allows reductions to be performed in any order, providing a simple model of program transformation and compiler optimization. The standard reduction property set forth in the Curry-Feys standardization theorem specifies a reduction sequence that only performs "necessary" reductions, providing a simple model for program execution.

We now establish call-by-need analogues of the Church-Rosser theorem and the Curry-Feys standardization theorem for classical λ-calculus.

Theorem 4.1. λ_{let} is confluent:

$$\begin{tikzcd} & M \arrow[dl, twoheadrightarrow, "\mathsf{let}"] \arrow[dr, twoheadrightarrow, "\mathsf{let}"'] & \\ M' \arrow[dr, dotted, twoheadrightarrow, "\mathsf{let}"'] & & M'' \arrow[dl, dotted, twoheadrightarrow, "\mathsf{let}"] \\ & \exists N & \end{tikzcd}$$

The full proof is given in the appendix, cumulating in the theorem Theorem A.19.

Proof Sketch: We first show that the system consisting of just let-I and let-V is confluent, using Plotkin's method of parallel reductions [Plo75]. We then show that the remaining reductions let-C, let-A and let-GC are both weakly Church-Rosser and strongly normalizing, and thus Church-Rosser. Since both subsystems commute, the theorem follows from the Lemma of Hindley and Rosen [Bar81, Proposition 3.3.5]. $\square$

The confluence result shows that different orders of reduction cannot yield different normal forms. It still might be the case that some reduction sequences terminate with a normal form while others do not terminate at all. However, the notion of reduction can be restricted to a *standard reduction* that always reaches an answer if there is one.

Figure 3 details our notion of standard reduction. To state the standard reduction property, we first make precise the kind of observations that can be made about λ_{let} programs. Following the spirit of [Abr90], we define an observation to be a reduction

Additional Syntactic Domains

Standard Values	V_s	$::=$	$\lambda x.M$
Answers	A, A'	$::=$	$V_s \mid \mathsf{let}\ x = M\ \mathsf{in}\ A$
Evaluation Contexts	E, E'	$::=$	$[\,] \mid E\ M \mid \mathsf{let}\ x = M\ \mathsf{in}\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E'[x]$

Standard Reduction Rules

(let_s-I)	$(\lambda x.M)\ N$	$\underset{s}{\Longrightarrow}$	$\mathsf{let}\ x = N\ \mathsf{in}\ M$
(let_s-V)	$\mathsf{let}\ x = V_s\ \mathsf{in}\ E[x]$	$\underset{s}{\Longrightarrow}$	$\mathsf{let}\ x = V_s\ \mathsf{in}\ E[V_s]$
(let_s-C)	$(\mathsf{let}\ x = L\ \mathsf{in}\ A)\ N$	$\underset{s}{\Longrightarrow}$	$\mathsf{let}\ x = L\ \mathsf{in}\ A\ N$
(let_s-A)	$\mathsf{let}\ y = (\mathsf{let}\ x = L\ \mathsf{in}\ A)\ \mathsf{in}\ E[y]$	$\underset{s}{\Longrightarrow}$	$\mathsf{let}\ x = L\ \mathsf{in}\ \mathsf{let}\ y = A\ \mathsf{in}\ E[y]$

Figure 3: Standard call-by-need reduction.

sequence that ends in a function term. In λ_{let}, a function term can be wrapped in let-bindings (since those can be pulled out of a function application by rule let-C). Hence, an answer A is either an abstraction or a let with an answer as its body.

Standard reduction is a restriction of ordinary reduction in that each redex must occupy the hole of an evaluation context. The first two productions for evaluation contexts in Figure 3 are those of the call-by-name calculus. The third production states that evaluation is possible in the body of a let. The final production emphasizes the call-by-need aspect of the strategy. It says that a definition should be evaluated if the defined node is demanded (*i.e.*, it appears in evaluation position itself).

The restriction to evaluation contexts alone does not make call-by-need reduction deterministic. For instance,

$$\mathsf{let}\ x = V\ \mathsf{in}\ \mathsf{let}\ y = W\ \mathsf{in}\ x\ y$$

has both let's in evaluation position, and hence would admit either the substitution of V for x or the substitution of W for y. We arrive at a deterministic standard reduction by specializing reduction rules to those shown in the second half of Figure 3. Note that there is no rule let_s-GC, since garbage-collection is not needed to reduce a term to an answer.

Definition. Let $\underset{s}{\Longrightarrow}$ be the smallest relation that contains let_s-{IVCA} and that is closed under the implication $M \underset{s}{\Longrightarrow} N \ \Rightarrow\ E[M] \underset{s}{\Longrightarrow} E[N]$.

Definition. We write $M \Downarrow$ iff there is an answer A such that $M \twoheadrightarrow A$. Likewise, we write $M \Downarrow_s$ iff there is an answer A such that $M \underset{s}{\Longrightarrow\!\!\!\!\!\rightarrow} A$.

Theorem 4.2. $\underset{s}{\Longrightarrow}$ is a standard reduction relation for λ_{let}. For all $M \in \Lambda$,

$$M \Downarrow\ \Leftrightarrow\ M \Downarrow_s\ .$$

Proof Sketch: The proof relies on two subsidiary results: that all answers are in $\underset{s}{\Leftrightarrow}$-normal form, and that standard reduction keeps head and internal redexes separate [Bar81, Lemma 11.4.3]. The result then follows by reasoning as in Barendregt's standardization proof for the call-by-name calculus. $\square$

5 Observational Equivalence

The call-by-need calculus is confluent and has a standard reduction order, and so it is, at least, a workable calculus. As of yet, though, we have to explore the relationship between λ_{let} and λ. The conversion theories $=_{\lambda_{\mathsf{let}}}$ and $=_{\lambda}$ are clearly different — otherwise there would be little point in studying call-by-need systems! However, we will show that the observational equivalence theories of λ and λ_{let} coincide on their common term language, Λ.

To keep the different equational theories of λ and λ_{let} apart, we will prefix reductions and convergence statements by the theory in which they are valid, *e.g.*, $\lambda{*} \vdash M \Downarrow$. Here and in the following, $\lambda{*}$ stands for either λ or λ_{let}.

A function term in λ is just a λ-abstraction. Hence, answers in λ are taken to be λ-abstractions and we define:

Definition. $\lambda \vdash M \Downarrow$ iff there is an abstraction $\lambda x.N$ such that $\lambda \vdash M \twoheadrightarrow \lambda x.N$.

Observational equivalence is the coarsest equivalence relation over terms that still distinguishes between terms with different observational behavior. Formally:

Definition. Two terms $M, N \in \Lambda{*}$ are *observationally equivalent* in $\lambda{*}$, written $\lambda{*} \models M \cong N$, iff for all $\lambda{*}$-contexts C such that $C[M]$ and $C[N]$ are closed,

$$\lambda{*} \vdash C[M] \Downarrow \quad \Leftrightarrow \quad \lambda{*} \vdash C[N] \Downarrow .$$

The remainder of this section works toward a theorem that the observational equivalence theories of λ and λ_{let} coincide on Λ. The first step towards this goal is Proposition 5.8, which links a term in λ_{let} with its let-expanded version in λ.

Definition. The let-expansion M^* of a term $M \in \Lambda_{\mathsf{let}}$ is defined inductively as follows:

$$\begin{array}{lcl}
x^* & = & x \\
(\lambda x.M)^* & = & \lambda x.M^* \\
(M\,N)^* & = & M^*\,N^* \\
(\mathsf{let}\ x = M\ \mathsf{in}\ N)^* & = & [M^*/x]\,N^*
\end{array}$$

let-expansion extends to contexts by $[\,]^* = [\,]$.

The following proposition is a direct consequence of the finite developments theorem in λ-calculus (with let terms as labeled β-redexes).

Proposition 5.1. Every term in Λ_{let} has a let-expansion.

Lemma 5.2. For all terms M, N,

$$[N^*/x]\, M^* \equiv ([N/x]\, M)^*.$$

Proof: By an easy structural induction on the form of M. □

Lemma 5.3. Let C be a single-hole context in λ_{let} such that C^* is an n-hole context ($n \geq 0$). Then is a substitution σ such that for all terms $M \in \Lambda_{\mathsf{let}}$

$$(C[M])^* \;\equiv\; C^*[\underbrace{\sigma M^*, ..., \sigma M^*}_{n}]$$

Proof: Let x_i $(i = 1, ..., m)$ be the variables that are let-bound in C and that have a bound occurrence in $C[x_i]$ and let N_i $(i = 1, ..., m)$ be their defining terms. Then by the definition of let-expansion the lemma holds with $\sigma = [N_i^*/x_i]_{i=1,...,m}$. □

Lemma 5.4. For all terms M, N, if $M \to N$ by a let-V-C-A-GC reduction, then $M^* \equiv N^*$.

Proof: A straightforward analysis of reduction rules. □

Lemma 5.5. For all terms $M, N \in \Lambda_{\mathsf{let}}$ the following diagram commutes.

$$\begin{tikzcd}
M \arrow[r, "\mathsf{let}"'] \arrow[d, "*"] & N \arrow[d, "*"'] \\
M^* \arrow[r, two heads, "\beta"'] & N^*
\end{tikzcd} .$$

Proof: By a structural induction on the proof of reduction $\underset{\mathsf{let}}{\twoheadrightarrow}$. For let-V-C-A-GC reductions at top-level, the lemma follows from Lemma 5.4. For toplevel let-I reductions, observe that

$$((\lambda x.M)N)^* \;\equiv\; (\lambda x.M^*)N^* \;\underset{\beta}{\twoheadrightarrow}\; [N^*/x]M^* \;\equiv\; ([N/x]M)^*$$

where the last equivalence follows from Lemma 5.2. It remains to show the lemma for the case where the last step in the proof of reduction is

$$\frac{M \underset{\mathsf{let}}{\twoheadrightarrow} N}{C[M] \underset{\mathsf{let}}{\twoheadrightarrow} C[N]}$$

for some terms M, N, context C. By Lemma 5.3 there is an integer $n \geq 0$ and a substitution σ such that

$$\begin{aligned}
(C[M])^* \;&\equiv\; C^*[\underbrace{\sigma M^*, ..., \sigma M^*}_{n}] \\
(C[N])^* \;&\equiv\; C^*[\underbrace{\sigma N^*, ..., \sigma N^*}_{n}]\ .
\end{aligned}$$

By the induction hypothesis, $M^* \underset{\beta}{\twoheadrightarrow} N^*$. Therefore, $(C[M])^* \;\equiv\; C^*[\sigma M^*, ..., \sigma M^*] \;\underset{\beta}{\twoheadrightarrow}\; C^*[\sigma N^*, ..., \sigma N^*] \;\equiv\; (C[N])^*$. □

For the proof of the next lemmas we need some more notation and definitions. In the following we will write a term $\mathsf{let}\ x_1 = M_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = M_n\ \mathsf{in}\ M$ alternatively as $\mathsf{let}\ \Phi\ \mathsf{in}\ M$, where the *heap* Φ is the sequence of bindings $(x_1 \mapsto M_1, ..., x_n \mapsto M_n)$[2]. In general, heaps are finite sequences of bindings $(x_i \mapsto M_i)_i$ such that $x_i = x_j \Rightarrow i = j$ and $x_i \in \mathrm{fv}(M_j) \Rightarrow i < j$. Let (Φ_1, Φ_2) denote the concatenation of the heaps Φ_1 and Φ_2 where is assumed that the bound variables of Φ_1 and Φ_2 form disjoint sets. Furthermore, let the *defining occurence* of a variable x in a heap Φ be

$$\mathrm{def}(x, \Phi) \quad = \quad \begin{cases} \mathrm{def}(y, \Phi) & \text{if } x \mapsto y \in \Phi, \\ M & \text{if } x \mapsto M \in \Phi \text{ and } M \text{ is not a variable}, \\ x & \text{otherwise.} \end{cases}$$

Finally, we extend let-expansion to a mapping from heaps to substitutions, by defining

$$(x_1 \mapsto M_1, ..., x_n \mapsto M_n)^* \quad = \quad [M_1^*/x_1]...[M_n^*/x_n] \ .$$

Lemma 5.6. For all terms $M \in \Lambda_{\mathsf{let}}$, $N \in \Lambda$ there exist terms $M' \in \Lambda_{\mathsf{let}}$, $N' \in \Lambda$ such that the following diagram commutes.

$$\begin{tikzcd} M \arrow[rr, dotted, "\mathsf{let}"'] \arrow[d, "*"] & & M' \arrow[d, "*"'] \\ M^* \arrow[r, "\beta"'] & N \arrow[r, dotted, twoheadrightarrow, "\beta"'] & N' \end{tikzcd} \quad .$$

Proof: W.l.o.g. assume that the term M in this diagram is in let-V-C-A-GC normal form — if it is not we can can always reduce it to such a normal form (Lemma A.15), and the reduction keeps the image under $(^*)$ invariant (Lemma 5.4).

We use a structural induction on the proof of $M^* \xrightarrow[\beta_s]{} N$. There are two cases.

Case 1 The reduction is a top-level application of the β rule. In this case,

$$M^* \equiv (\lambda x.Q_1)Q_2 \xrightarrow[\beta_s]{} [Q_1/x]Q_2 \equiv N$$

for some variable x, terms Q_1, Q_2. Let $M \equiv \mathsf{let}\ \Phi\ \mathsf{in}\ M_0$ where M_0 is not a let-binding. Then one of the following subcases applies.

Case 1.1 $M_0 \equiv z$, for some variable z and $\mathrm{def}(z, \Phi) = (\lambda x.P_1)P_2$ for some terms P_1, P_2 with $\Phi^* P_i^* \equiv Q_i^*$ $(i = 1, 2)$. By the definition of $\mathrm{def}(\_,\_)$ there exist heaps Φ_1, Φ_2 and a variable y such that $\Phi = (\Phi_1, y \mapsto (\lambda x.P_1)P_2, \Phi_2)$ and $\mathrm{def}(z, \Phi_2) = y$. Therefore,

$$\begin{array}{lll} M & \equiv & \mathsf{let}\ \Phi_1, y \mapsto (\lambda x.P_1)P_2, \Phi_2\ \mathsf{in}\ z \\ & \xrightarrow[\mathsf{let\text{-}I}]{} & \mathsf{let}\ \Phi_1, y \mapsto \mathsf{let}\ x = P_2\ \mathsf{in}\ P_1, \Phi_2\ \mathsf{in}\ z \\ & \xrightarrow[\mathsf{let\text{-}A}]{} & \mathsf{let}\ \Phi_1, x \mapsto P_2, y \mapsto P_1, \Phi_2\ \mathsf{in}\ z \qquad \stackrel{\mathrm{def}}{=} \quad M' \end{array}$$

[2]Note the relationship to the natural semantics in Section 7.

Furthermore,

$$\begin{array}{rcll}(M')^* & \equiv & \Phi_1^*([P_2^*/x]([P_1^*/y](\Phi_2^* z))) & \\ & \equiv & \Phi_1^*([P_2^*/x]([P_1^*/y]y)) & \text{since } \mathrm{def}(z,\Phi_2) = y \\ & \equiv & \Phi_1^*([P_2^*/x]P_1^*) & \\ & \equiv & \Phi^*([P_2^*/x]P_1^*) & \text{since } \mathrm{fv}([P_2^*/x]P_1^*) \cap (\mathrm{dom}(\Phi_2) \cup \{x,y\}) = \emptyset \\ & \equiv & [Q_2/x]Q_1 & \text{by Lemma 5.2 .}\end{array}$$

Case 1.2 $M_0 \equiv (\lambda x.P_1)P_2$, for some terms P_1, P_2 with $\Phi^* P_i^* \equiv Q_i^*$ $(i = 1, 2)$. This case is similar, but somewhat simpler, than the previous case. It is omitted here.

No other subcases apply. To see this, assume that $M_0 \equiv \lambda y.P$, for some variable y, term P. Then by definition of $(^*)$, $M^* \equiv \lambda y.\Phi^* P^*$, a contradiction. Or assume that $M_0 \equiv P_1P_2P_3$ for some terms P_1, P_2, P_3. In this case, $M^* \equiv (\Phi^* P_1^*)(\Phi^* P_2^*)(\Phi^* P_3^*)$, a contradiction. Finally, assume that $M_0 \equiv yP$, for some variable y, term P. We then distinguish according to $\mathrm{def}(y, \Phi)$. If $\mathrm{def}(y, \Phi)$ is a variable, say z, we get $M^* \equiv z(\Phi^* P^*)$, a contradiction. If $\mathrm{def}(y, \Phi)$ is a λ-abstraction we get a contradiction since then M is not in let-V-C-A-GC normal form. Finally, if y is an application P_1P_2 we get a contradiction since in that case $M^* \equiv (\Phi^* P_1^*)(\Phi^* P_2^*)(\Phi^* P_3^*)$.

This concludes the case where the reduction $M^* \underset{s}{\Leftrightarrow} N$ is top-level.

Case 2 The redex in the reduction $M^* \underset{s}{\Leftrightarrow} N$ is a proper subterm of M^*. In this case the last step in the proof of this reduction is an application of the rule

$$\frac{Q_1 \underset{s}{\Leftrightarrow} Q_1'}{Q_1Q_2 \underset{s}{\Leftrightarrow} Q_1'Q_2}$$

where $M^* \equiv Q_1Q_2$. Let again $M \equiv \mathsf{let}\ \Phi\ \mathsf{in}\ M_0$ where M_0 is not a let-binding. Then one of the following cases applies.

Case 1.1 $M_0 \equiv z$, for some variable z and $\mathrm{def}(z, \Phi) = P_1P_2$, for some terms P_1, P_2 with $\Phi^* P_i^* \equiv Q_i^*$ $(i = 1, 2)$. Then there exist heaps Φ_1, Φ_2 and a variable y such that $\Phi = (\Phi_1, y \mapsto P_1P_2, \Phi_2)$ and $\mathrm{def}(z, \Phi_2) = y$. Since $\mathrm{fv}(P_1, P_2) \cap (\mathrm{dom}(\Phi_2) \cup y) = \emptyset$, we have also that $\Phi_1^* P_i^* \equiv Q_i^*$ $(i = 1, 2)$. By the induction hypothesis, $\mathsf{let}\ \Phi_1\ \mathsf{in}\ P_1 \underset{\mathsf{let}}{\Leftrightarrow\!\!\!\twoheadrightarrow} P_1'$, for some term P_1' such that $Q_1' \underset{\beta}{\Leftrightarrow\!\!\!\twoheadrightarrow} (P_1')^*$. A straightforward induction on the notion of reduction in λ_{let} shows that P_1' is of the form $\mathsf{let}\ \Phi_1'\ \mathsf{in}\ P_1''$ where for all terms P, $\mathsf{let}\ \Phi_1\ \mathsf{in}\ P \underset{\mathsf{let}}{\Leftrightarrow\!\!\!\twoheadrightarrow} \mathsf{let}\ \Phi_1'\ \mathsf{in}\ P$. It follows that

$$\begin{array}{rcl}M & \equiv & \mathsf{let}\ \Phi_1, y \mapsto P_1P_2, \Phi_2\ \mathsf{in}\ z \\ & \underset{\mathsf{let}}{\Leftrightarrow\!\!\!\twoheadrightarrow} & \mathsf{let}\ \Phi_1'\ y \mapsto P_1''P_2, \Phi_2\ \mathsf{in}\ z \ \stackrel{\mathrm{def}}{=}\ M' \ .\end{array}$$

Furthermore,

$$\begin{array}{rcll}
(M')^* & \equiv & (\Phi_1')^*([(P_1''P_2)^*/y](\Phi_2^* z)) & \\
& \equiv & (\Phi_1')^*([(P_1''P_2)^*/y]y) & \text{since } \mathrm{def}(z, \Phi_2) = y \\
& \equiv & ((\Phi_1')^*(P_1'')^*)((\Phi_1')^* P_2^*) & \\
& \equiv & Q_1'((\Phi_1')^* P_2^*) & \\
& \stackrel{\mathrm{def}}{\equiv} & N' \ . &
\end{array}$$

Since $(\Phi_1')^* P_2^* \equiv (\mathsf{let}\ \Phi_1'\ \mathsf{in}\ P_2)^*$ and $\mathsf{let}\ \Phi_1\ \mathsf{in}\ P_2 \twoheadrightarrow \mathsf{let}\ \Phi_1'\ \mathsf{in}\ P_2$, it follows with Lemma 5.5 that $Q_2 \equiv (\mathsf{let}\ \Phi_1\ \mathsf{in}\ P_2)^* \underset{\beta}{\Leftrightarrow\!\!\!\twoheadrightarrow} (\Phi_1')^* P_2^*$. In summary, $Q_1' Q_2 \underset{\beta}{\Leftrightarrow\!\!\!\twoheadrightarrow} N'$.

Case 2.2 $M_0 \equiv P_1 P_2$, for some terms P_1, P_2 with $\Phi^* P_i^* \equiv Q_i^*$ $(i = 1, 2)$. This case is similar, but somewhat simpler, than the previous case. It is omitted here.

By a similar reasoning as for Case 1, we can show that no other subcases apply. □

We also need a fact about standard reduction in classical lambda calculus.

Notation We write $M \rightarrow^n N$ if M reduces in at most n steps to N.

Proposition 5.7. Let $M, N \in \Lambda$, such that $\lambda \vdash M \underset{i}{\Leftrightarrow\!\!\!\twoheadrightarrow} N$ and $\lambda \vdash M \underset{s}{\Leftrightarrow\!\!\!\twoheadrightarrow}^n A$, for some answer A, $n \geq 0$. Then $N \underset{s}{\Leftrightarrow\!\!\!\twoheadrightarrow}^n A'$ for some answer A'.

The proof of Proposition 5.7 is found in Appendix A.3. Now everything is in place for our central technical result.

Proposition 5.8. For all $M \in \Lambda_{\mathsf{let}}$,

$$\lambda_{\mathsf{let}} \vdash M \Downarrow \quad \Leftrightarrow \quad \lambda \vdash M^* \Downarrow .$$

Proof: "$\Rightarrow$": An easy induction on the length of reduction from M to an answer, using Lemma 5.5 at each step.

"$\Leftarrow$": Assume that $M^* \Downarrow$. By the Curry-Feys standardisation theorem [Bar81, 11.4.7], M reduces to an answer by a sequence of standard reductions. We use an induction on the length n of this sequence. If $n = 0$ then M^* is an answer, i.e. a λ-abstraction, say $\lambda x.Q$. Then the definition of let-expansion implies that one of two cases apply. Either $M \equiv \mathsf{let}\ \Phi\ \mathsf{in}\ \lambda x.P$, for some heap Φ, term P such that $\Phi^* P^* \equiv Q$. Then M is an answer in λ_{let}. Or $M \equiv \mathsf{let}\ \Phi\ \mathsf{in}\ z$, for some heap Φ, variable z such that $\mathrm{def}(z, \Phi) = \lambda x.P$ and $\Phi^* P^* \equiv Q$. In that case M reduces to the answer $\mathsf{let}\ \Phi\ \mathsf{in}\ \lambda x.P$ by a sequence of let-V reductions.

Assume now that $M^* \Downarrow_s^n$ for $n > 0$. Let N be the standard reduct of M^*, i.e. $M^* \underset{s}{\Leftrightarrow\!\!\!\twoheadrightarrow} N$ and $N \Downarrow_s^{n-1}$. By Lemma 5.6 there is a term $M' \in \Lambda_{\mathsf{let}}$ such that $\lambda_{\mathsf{let}} \vdash M \twoheadrightarrow M'$ and $N \underset{\beta}{\Leftrightarrow\!\!\!\twoheadrightarrow} (M')^*$. By Proposition 5.7, $(M')^* \Downarrow_s^{n-1}$, Therefore, by the induction hypothesis, $M' \Downarrow$. With $\lambda_{\mathsf{let}} \vdash M \twoheadrightarrow M'$ this implies that $M \Downarrow$. □

Proposition 5.8 implies that λ_{let} is a conservative observational extension of λ:

Theorem 5.9. The observational equivalence theories of λ and λ_{let} coincide on Λ. For all terms $M, N \in \Lambda$,

$$\lambda \models M \cong N \;\Leftrightarrow\; \lambda_{\mathsf{let}} \models M \cong N \;.$$

Proof: "$\Rightarrow$": Assume $\lambda \models M \cong N$ and let C be a λ_{let}-context such that $C[M]$ and $C[N]$ are closed. Let $C^{\#}$ result from C by eliminating all let's in C using rule let-I repeatedly in reverse. Then

$$\begin{array}{lll}
 & \lambda_{\mathsf{let}} \vdash C[M]\Downarrow & \\
\Leftrightarrow & \lambda_{\mathsf{let}} \vdash C^{\#}[M]\Downarrow & \text{since } \lambda_{\mathsf{let}} \vdash C^{\#}[M] = C[M] \\
\Leftrightarrow & \lambda \vdash C^{\#}[M]\Downarrow & \text{by Proposition 5.8, since } (C^{\#}[M])^{*} = C^{\#}[M] \\
\Leftrightarrow & \lambda \vdash C^{\#}[N]\Downarrow & \text{since } \lambda \models M \cong N \\
\Leftrightarrow & \lambda_{\mathsf{let}} \vdash C[N]\Downarrow & \text{by the reverse argument}
\end{array}$$

"$\Leftarrow$": By a symmetric argument, with C instead of $C^{\#}$, and leaving out the first step in the equivalence chain. □

Corollary 5.10. β is an observational equivalence in λ_{let}: For all $M, N \in \Lambda_{\mathsf{let}}$, $(\lambda x.M)N \cong [M/x]N$.

Proof: Let $M, N \in \Lambda_{\mathsf{let}}$. Let M', N' be the corresponding Λ-terms that result from eliminating all let's in M, N by performing let-I reductions in reverse. Then we have in λ_{let}:

$$(\lambda x.M)N \;=\; (\lambda x.M')N' \;\cong\; [N'/x]M' \;=\; [N/x]M$$

where "$\cong$" follows from Theorem 5.9. □

6 The Let-Less Call-By-Need Calculus

In call-by-name λ-calculus, $\mathsf{let}\ x = M\ \mathsf{in}\ N$ is syntactic sugar for $(\lambda x.N)\ M$: so let-bindings are not really essential. It turns out that we can use the same same expansion to get rid of let's in call-by-need. The resulting calculus is λ_ℓ (where the ℓ stands for "lazy"). Its notions of general and standard reduction are shown in Figure 4.

While λ_ℓ is perhaps somewhat less intuitive than λ_{let}, its simpler syntax makes some of the basic (syntactic) results easier to derive. It also allows better comparison with the call-by-name calculus, since no additional syntactic constructs are introduced.

Clearly, λ_{let} and λ_ℓ are closely related. More precisely, the following theorem states that reduction in λ_{let} can be simulated in λ_ℓ, and that the converse is also true, provided we identify terms that are equal up to let-I introduction.

Proposition 6.1. For all $M \in \Lambda_\ell$, $M' \in \Lambda_{\mathsf{let}}$,

Syntactic Domains

Variables	x, y, z		
Standard Values	V_s	$::=$	$\lambda x.M$
Values	V, W	$::=$	$x \mid V_s$
Terms	L, M, N	$::=$	$V \mid M\ N$
Answers	A, A'	$::=$	$V_s \mid (\lambda x.A)\ M$
Evaluation Contexts	E, E'	$::=$	$[\,] \mid E\ M \mid (\lambda x.E)\ M \mid (\lambda x.E'[x])\ E$

General Reduction Rules

$$
\begin{array}{llcll}
(\ell\text{-V}) & (\lambda x.C[x])\,V & \to & (\lambda x.C[V])\,V & \\
(\ell\text{-C}) & (\lambda x.L)MN & \to & (\lambda x.LN)M & \\
(\ell\text{-A}) & (\lambda x.L)((\lambda y.M)N) & \to & (\lambda y.(\lambda x.L)M)N & \\
(\ell\text{-GC}) & (\lambda x.M)\ N & \to & M & \textbf{if } x \notin \mathrm{fv}(M)
\end{array}
$$

Standard Reduction Rules

$$
\begin{array}{llcl}
(\ell_s\text{-V}) & (\lambda x.E[x])\,V_s & \underset{s}{\Longmapsto} & (\lambda x.E[V_s])\,V_s \\
(\ell_s\text{-C}) & (\lambda x.A)MN & \underset{s}{\Longmapsto} & (\lambda x.AN)M \\
(\ell_s\text{-A}) & (\lambda x.E[x])((\lambda y.A)N) & \underset{s}{\Longmapsto} & (\lambda y.(\lambda x.E[x])A)N
\end{array}
$$

Figure 4: The let-less call-by-need calculus.

$$
\begin{tikzcd}
M \arrow[r, two heads, "\ell"'] \arrow[d, "\mathsf{let}\text{-I}"'] & N \arrow[d, dotted, "\mathsf{let}\text{-I}"] \\
M' \arrow[r, dotted, two heads, "\mathsf{let}"'] & N'
\end{tikzcd}
\qquad
\begin{tikzcd}
M \arrow[r, dotted, two heads, "\ell"'] \arrow[d, "\mathsf{let}\text{-I}"'] & N \arrow[d, dotted, "\mathsf{let}\text{-I}"] \\
M' \arrow[r, two heads, "\mathsf{let}"'] & N'
\end{tikzcd}
\quad .
$$

Proposition 6.1 can be used to derive the essential syntactic and properties of λ_ℓ from those of λ_{let}:

Theorem 6.2. λ_ℓ is Church Rosser.

Theorem 6.3. $\underset{s}{\Longmapsto}$ is a standard reduction relation for λ_ℓ. For all $M \in \Lambda$,

$$M \Downarrow \;\Leftrightarrow\; M \Downarrow_s \quad .$$

λ_ℓ has close relations to both the call-by-value calculus λ_V and the call-by-name calculus λ. Its notion of equality $=_{\lambda_\ell}$ — *i.e.*, the least equivalence relation generated by the reduction relation — fits between those of the other two calculi, making λ_ℓ an extension of λ_V and λ an extension of λ_ℓ.

Theorem 6.4.

$$=_{\lambda_V} \;\subset\; =_{\lambda_\ell} \;\subset\; =_\lambda \quad .$$

Proof: (1) βV can be expressed by a sequence of λ_ℓ reductions as was shown at the end of Section 3. Therefore, $=_{\lambda_V} \subset =_{\lambda_\ell}$. (2) Each λ_ℓ reduction rule is an equality in λ. For instance, in the case of ℓ-V one has:

$$(\lambda x.C[x])\,V \quad =_\beta \quad [V/x](C[x]) \;\equiv\; [V/x](C[V]) \quad =_\beta \quad (\lambda x.C[V])\,V$$

The other rules have equally simple translations. In summary, $=_{\lambda_\ell} \subset =_\lambda$. □

Each of the inclusions of Theorem 6.4 is proper; *e.g.*,

$$(\lambda x.x)\,((\lambda y.y)\,\Omega) \quad = \quad (\lambda y.(\lambda x.x)\,y)\,\Omega$$

is an instance of rule ℓ-A, but it is not an equality in the call-by-value calculus (Ω stands for a non-terminating computation). On the other hand, the following instance of β is not an equality in λ_ℓ:

$$(\lambda x.x)\,\Omega \quad = \quad \Omega \;\; .$$

However, one can show by a simple application of Theorem 5.9 together with Proposition 6.1 that the observational equivalence theories of λ_ℓ and λ are identical (and are incompatible with the observational equivalence theory of λ_V).

Theorem 6.5. For all terms $M, N \in \Lambda$,

$$\lambda \models M \cong N \quad \Leftrightarrow \quad \lambda_\ell \models M \cong N.$$

Theorem 6.4 implies that any model of call-by-name λ-calculus is also a model of λ_ℓ, since it validates all equalities in λ_ℓ. Theorem 6.5 implies that any adequate (respectively fully-abstract) model of λ is also adequate (fully-abstract) for λ_ℓ, since the observational equivalence theories of both calculi are the same. For instance, Abramsky and Ong's model of the lazy lambda calculus [Abr90] is adequate for λ_ℓ.

7 Natural semantics

This section presents an operational semantics for call-by-need in the natural semantics style of Plotkin and Kahn, similar to one given by Launchbury [Lau93]. A proposition is stated that relates the natural semantics to standard reduction.

A *heap* abstracts the state of the store at a point in the computation. It consists of a sequence of pairs binding variables to terms,

$$x_1 \mapsto M_1,\; \ldots,\; x_n \mapsto M_n.$$

The order of the sequence of bindings is significant: all free variables of a term must be bound to the left of it. Furthermore, all variables bound by the heap must be distinct.

$$\text{Id}\ \frac{\langle \Phi \rangle\, M \Downarrow \langle \Psi \rangle\, V}{\langle \Phi,\, x \mapsto M,\, \Upsilon \rangle\, x \Downarrow \langle \Psi,\, x \mapsto V,\, \Upsilon \rangle\, V}$$

$$\text{Abs}\ \frac{}{\langle \Phi \rangle\, \lambda x.\, N \Downarrow \langle \Phi \rangle\, \lambda x.\, N}$$

$$\text{App}\ \frac{\langle \Phi \rangle\, L \Downarrow \langle \Psi \rangle\, \lambda x.\, N \qquad \langle \Psi,\, x' \mapsto M \rangle\, [x'/x]N \Downarrow \langle \Upsilon \rangle\, V}{\langle \Phi \rangle\, L\ M \Downarrow \langle \Upsilon \rangle\, V}$$

Figure 5: Operational semantics of call-by-need lambda calculus.

Thus the heap above is well-formed if $\text{fv}(M_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ for each i in the range $1 \leq i \leq n$, and all the x_i are distinct. Let Φ, Ψ, Υ range over heaps. If Φ is the heap $x_1 \mapsto M_1, \ldots, x_n \mapsto M_n$, define $\text{vars}(\Phi) = \{x_1, \ldots, x_n\}$. A configuration pairs a heap with a term, where the free variables of the term are bound by the heap. Thus $\langle \Phi \rangle\, M$ is well-formed if Φ is well-formed and $\text{fv}(M) \subseteq \text{vars}(\Phi)$. The operation of evaluation takes configurations into configurations. The term of the final configuration is always a value. Thus evaluation judgments take the form $\langle \Phi \rangle\, M \Downarrow \langle \Psi \rangle\, V$.

The rules defining evaluation are given in Figure 5. There are three rules, for identifiers, abstractions and applications.

- Abstractions are trivial. As abstractions are already values, the heap is left unchanged and the abstraction is returned.

- Applications are straightforward. Evaluate the function to yield a lambda abstraction, extend the heap so that the the bound variable of the abstraction is bound to the argument, then evaluate the body of the abstraction. In this rule, x' is a new name not appearing in Ψ or N. The renaming guarantees that each identifier in the heap is unique.

- Variables are more subtle. The basic idea is straightforward: find the term bound to the variable in the heap, evaluate the term, then update the heap to bind the variable to the resulting value. But some care is required to ensure that the heap remains well-formed. The original heap is partitioned into $\Phi,\, x \mapsto M,\, \Upsilon$. Since the heap is well-formed, only Φ is required to evaluate M. Evaluation yields a new heap Ψ and value V. The new heap Ψ will differ from the old heap Φ in two ways: binding may be updated (by Var) and bindings may be added (by App). The free variables of V are bound by Ψ, so to ensure the heap stays well-formed, the final heap has the form $\Psi,\, x \mapsto V,\, \Upsilon$.

A semantics of let terms can be derived from the above rules: the semantics of $\mathsf{let}\ x = M\ \mathsf{in}\ N$ is identical to the semantics of $(\lambda x.\, M)\, N$.

As one would expect, evaluation uses only well-formed configurations, and evaluation only extends the heap.

Syntactic Domains

$$
\begin{array}{llcll}
\text{Operators} & p & & & \\
\text{Constructors} & k^n & & \text{(of arity } n\text{)} & \\
\text{Values} & V, W & ::= & x \mid \lambda x.M \mid k^n\, V_1 \ldots V_n & (n \geq 0) \\
\text{Terms} & L, M, N & ::= & V \mid M\ N \mid \mathsf{let}\ x = M\ \mathsf{in}\ N \mid p &
\end{array}
$$

Additional Reduction Rules

$$
\begin{array}{llcll}
(\delta\text{-V}) & p\, V & \rightarrow & \delta(p, V) & (\delta(f, V) \text{ defined}) \\
(\delta\text{-A}) & p\, (\mathsf{let}\ x = M\ \mathsf{in}\ N) & \rightarrow & \mathsf{let}\ x = M\ \mathsf{in}\ p\ N &
\end{array}
$$

Figure 6: Data constructors and primitive operations.

Lemma 7.1. Given an evaluation tree with root $\langle\Phi\rangle\, M \Downarrow \langle\Psi\rangle\, V$, if $\langle\Phi\rangle\, M$ is well-formed then every configuration in the tree is well-formed, and furthermore $\mathrm{vars}(\Phi) \subseteq \mathrm{vars}(\Psi)$.

Thanks to the care taken to preserve the ordering of heaps, it is possible to draw a close correspondence between evaluation and standard reductions. If Φ is the heap $x_1 \mapsto M_1, \ldots, x_n \mapsto M_n$, write $\mathsf{let}\ \Phi\ \mathsf{in}\ N$ for the term

$$\mathsf{let}\ x_1 = M_1\ \mathsf{in}\ \cdots\ \mathsf{let}\ x_n = M_n\ \mathsf{in}\ N.$$

Every answer A can be written $\mathsf{let}\ \Psi\ \mathsf{in}\ V$ for some heap Ψ and value V. Then a simple induction on $\Downarrow$-derivations yields the following result.

Proposition 7.2. $\langle\Phi\rangle\, M \Downarrow \langle\Psi\rangle\, V$ if and only if $\lambda_\ell \vdash \mathsf{let}\ \Phi\ \mathsf{in}\ M \twoheadrightarrow_{s} \mathsf{let}\ \Psi\ \mathsf{in}\ V$.

The semantics given here is similar to that presented by Launchbury [Lau93]. An advantage of our semantics over Launchbury's is that the form of terms is standard, and care is taken to preserve ordering in the heap. Launchbury uses a non-standard syntax, in order to achieve a closer correspondence between terms and evaluations: in an application the argument to a term must be a variable, and all bound variables must be uniquely named. Here, general application is supported directly and all renaming occurs as part of the application rule. It is interesting to note that Launchbury presents an alternative formulation quite similar to ours, buried in one of his proofs.

An advantage of Launchbury's semantics over ours is that his copes more neatly with recursion, by the use of multiple, recursive let bindings. An extension of our semantics to include recursion (such as that of Ariola and Felleisen [AF94]) would lose the ordering property of the heap, and hence lose the close connection to standard reductions [WT94].

8 Extensions

Functional programming languages generally have more constructs than just function abstraction and application. Typically, data constructors and selectors as well as various other primitive operators are provided. Of course, these additions can be simulated in the base language via Church encodings. Yet a more high-level treatment is often desirable for reasons of both clarity and efficiency. The full paper will detail how these extensions can be added to the call-by-need calculus.

Figure 6 extends λ_{let} with data constructors k^n of arbitrary arity n and primitive operators p (of which selectors are a special case). There is one new form of value: $k^n\ V_1\ \ldots\ V_n$ where the components $V_1;\ldots;V_n$ must be values — otherwise sharing would be lost when copying the compound value. For instance, $inl\ (1+1)$ is not a legal value, since copying it would also copy the unevaluated term $(1+1)$. Instead, one writes $\mathsf{let} x = 1+1\ \mathsf{in}\ inl\ x$.

There are two new reduction rules. Rule δ-V is the usual rewrite rule for primitive operator application. It is defined in terms of a partial function — also called δ — from operators and values to terms. This function can be arbitrary, as long as it does not "look inside" lambda abstractions. That is, we postulate that for all operators p and contexts C there is a context D such that for all terms M, $\delta(p, C[\lambda x.M]) = D[\lambda x.M]$ or $\delta(p, C[\lambda x.M])$ is undefined. Note that rule δ-V makes all primitive operators unary and strict. Operators of more than one argument can still be simulated by currying. Rule δ-A allows $\mathsf{let} \Leftrightarrow bindings$ of operator arguments to be pulled out of the application.

Modelling Heap Maintenance. The transition from the general call-by-need calculus to the standard scheme pares away steps not strictly needed for reduction to an answer. As we have seen, garbage collection is one sort of these steps.

Related to garbage collection — and performed by many implementation at the same time as garbage collection — is the task of *shorting out indirections* [Pey87]. An indirection node is a graph element that only points to some other graph node, and contains no other information itself. Since referencing through such indirection wastes time, pointers to indirection nodes should be replaced with pointers to what the indirection itself points to; although retaining indirection nodes decreases efficiency, their presence should not disrupt program evaluation.

In λ_ℓ, indirection shortening is a sort of non-standard ℓ-V redex: namely, the case where the argument is a single free variable.

$$\ell\text{-S}: \ (\lambda x.C[x])y \ \rightarrow \ [y/x]M \ .$$

Let λ_ℓ^S be the theory obtained by extending λ_ℓ with ℓ-S.

The extension does not alter the resulting theory:

Theorem 8.1.

$$\lambda_\ell^S \models M \cong N \quad \Leftrightarrow \quad \lambda_\ell \models M \cong N \ .$$

Proof: Trivial, by Theorem 5.9, since ℓ-S contractions are just ordinary β contractions. □

Likewise, a rule for indirection shortening will not alter the theory λ_{let}.

Although Launchbury does not identify an indirection shortening rule, such an extension for his system would be relatively simple, *viz.*,

$$\frac{[y/x],\ : e \ \Downarrow\ \Delta : z}{(,\ , x \mapsto y) : e \ \Downarrow\ \Delta : z},\ x \notin \mathrm{fv}(,\ )\ .$$

ℓ-GC and ℓ-S allow a nice, two-layered generalization of the standard ordering. At one layer is ℓ_s-{V,C,A}, by itself deterministic and which will therefore always reach answers when they exist. At a second layer, ℓ-{GC,S} reduction, while not deterministic, is strongly normalizing, and the two layers together form a confluent calculus which, like ℓ_s-{V,C,A}, will produce an answer whenever possible. Their interaction is exactly the role we expect a garbage collector to play: we may cease reduction at any time to collect as much garbage as we like without altering the eventual result.

9 Conclusion

The calculus presented here has several nice properties that make it suitable as a reasoning tool for lazy functional programs: it operates on the lambda-terms themselves — or possibly a mildly sugared version — rather than needing a separate store of bindings; it can be defined by a few simple rules; its theory extends to subterms, even those under abstractions. The calculus fits naturally between the call-by-value and call-by-name versions of λ. It shares with call-by-value the property that only values are copied, yet validates all observational equivalences of call-by-name.

A shortcoming of our approach is its treatment of recursion. We express recursion with a fixpoint combinator (which is definable since our calculus is untyped). This agrees with Wadsworth's original treatment and most subsequent formalizations of call-by-need[3]. However, implementations of lazy functional languages generally express recursion by a back-pointer in the function graph. The two schemes are equivalent for recursive function definitions but they have different sharing behavior in the case of circular data structures. A circular pointer can allow more efficient sharing in the case of (say)

$$\mathsf{let}\ xs\ =\ (1+1) : xs\ \mathsf{in}\ xs\ .$$

It seems possible to extend our calculus with a recursive let-construct on order to better model recursion. This remains a topic for future work.

Acknowledgements. The authors would like to thank Zena Ariola, Matthias Felleisen, John Field and David N. Turner for valuable discussions.

References

[Abr90] Samson Abramsky. *The Lazy Lambda Calculus*, chapter 4, pages 65–116. The UT Year of Programming Series. Addison-Wesley Publishing Company, Inc., 1990.

[3] with the notable exception of [Lau93].

[ACCL90] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Proc. 18th ACM Symposium on Principles of Programming Languages, San Francisco, California*. ACM Press, January 1990.

[AF94] Zena Ariola and Matthias Felleisen. The call-by-need lambda calculus. Technical Report CIS-TR-94-23, Department of Computer Science, University of Oregon, October 1994.

[AFM+95] Zena Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Phillip Wadler. A call-by-need lambda calculus. In *Proc. 22nd Sym. on Principles of Programming Languages, San Francisco*. ACM Press, January 1995.

[Bar81] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, 1981.

[Fie90] John Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proc. 18th ACM Symposium on Principles of Programming Languages, San Francisco, California*. ACM Press, January 1990.

[Joh84] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the 1984 ACM SIGPLAN Conference on Compiler Construction*, New York, June 1984. ACM.

[Jos89] Mark B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68:105–111, 1989.

[KL89] Philip J. Koopman Jr. and Peter Lee. A fresh look at combinator graph reduction. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*. ACM, ACM Press, June 1989.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *Proc. 21st ACM Symposium on Principles of Programming Languages, Charleston, South Carolina*. ACM Press, January 1993.

[Mar91] Luc Maranget. Optimal derivations in weak lambda-calculi and in orthogonal term rewriting systems. In *Proc. 19th ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 255–269. ACM Press, January 1991.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Ong88] Chih-Hao Luke Ong. Fully abstract models of the lazy lambda calculus. In *Proceedings of the 29th Symposium on Foundations of Computer Science*, pages 368–376. IEEE, 1988.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice Hall, 1987.

[Pey92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, July 1992.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PS92] S. Purushothaman and Jill Seaman. An adequate operational semantics of sharing in lazy evaluation. In B. Krieg-Brückner, editor, *Lecture Notes in Computer Science 582*, pages 435–450, New York, February 1992. ESOP'92, Fourth European Symposium on Programming, Springer-Verlag.

[SP94] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict higher-order functional language, July 1994. Submitted to *22'nd ACM Symposium on Principles of Programming Languages.*

[Tur79] David A. Turner. A new implementation technique for applicative programming languages. *Software—Practice and Experience*, 9(31–49), 1979.

[Wad71] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* PhD thesis, Oxford University, 1971.

[WT94] Philip Wadler and David N. Turner. Once upon a type. Presented to the Semantique Workshop, Århus, Denmark, June 1994.

[Yos93] Nobuko Yoshida. Optimal reduction in weak-lambda-calculus with shared environments. In *FPCA'93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, New York, June 1993. ACM Press.

A Proofs

A.1 Church-Rosser

Definition A.1. Let the set Λ^*_{let} be the collection all terms matching T in the grammar:

$$T \ ::= \ x^i \ \mid \ \lambda x.T \ \mid \ T\ T \ \mid \ \mathsf{let}\ x = T\ \mathsf{in}\ T$$

where i is a positive integer.

Let ι be a map from the individual variables of a term $T \in \Lambda_{\mathsf{let}}$ to positive integers; then (T, ι) denotes the Λ^*_{let} term obtained by weighting T's variables according to ι.

Define the projection map $|\,.\,| : \Lambda^*_{\mathsf{let}} \to \Lambda_{\mathsf{let}}$ by erasing the weights from a term.

Lemma A.2. For every term $T^* \in \Lambda^*_{\mathsf{let}}$, there exists some ι such that $T^* \equiv (|T^*|, \iota)$.

Proof: Trivial; each term's weights define the appropriate function. $\Box$

Definition A.3. The norm $\|\,.\,\| : \Lambda^*_{\mathsf{let}} \to N$, by

$$\begin{array}{rcl} \|x^i\| & = & i + 1 \\ \|e_1\ e_2\| & = & \|e_1\| \cdot \|e_2\| \\ \|\lambda x.e\| & = & 2 + \|e\| \\ \|\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\| & = & \|e_1\| \cdot \|e_2\| + 1\ \ . \end{array}$$

Lemma A.4. Let $T \in \Lambda^*_{\mathsf{let}}$. Then $\|T^*\| > 1$.

Proof: Trivial. □

Definition A.5. A term $T^* \in \Lambda^*_{\mathsf{let}}$ is said to *have a decreasing weighting* (or for a term (T, ι), that ι *is* a decreasing weighting) if for every subterm $\mathsf{let}\ x = M^*\ \mathsf{in}\ N^*$ of T, and for every x^i in N, we have

$$i > \|M\| \ .$$

Algorithm A.6. Given a term T, we first number the variable occurances in T from two from right to left, except that within an expression $\mathsf{let}\ x = M\ \mathsf{in}\ N$ we number M and then number N. Then we obtain a weighting for T by giving the variable numbered i the weighting i^i.

Lemma A.7. Every term has a decreasing weighting.

Proof: As assigned by the above algorithm, as in [Bar81, Lemma 11.2.17]. □

Lemma A.8. Let $T^* \in \Lambda^*_{\mathsf{let}}$ and

$$T^* \xrightarrow[\mathsf{let}\text{-}\{\mathrm{I,C,A,GC}\}]{} T^{*\prime} \ .$$

Then $\|T^{*\prime}\| < \|T^*\|$.

Proof: Trivial, by analysis of the before-and-after term structures. □

Lemma A.9. Let $T^* \equiv (T, \iota) \in \Lambda^*_{\mathsf{let}}$, ι a decreasing weighting for T, and

$$T^* \equiv (T, \iota) \xrightarrow[\mathsf{let}\text{-}\mathrm{V}]{} T^{*\prime} \ .$$

Then $\|T^{*\prime}\| < \|T^*\|$.

Proof: Again, by analysis of the terms. □

Corollary A.10. Let $T^* \equiv (T, \iota) \in \Lambda^*_{\mathsf{let}}$, ι a decreasing weighting for T, and

$$T^* \equiv (T, \iota) \xrightarrow[\lambda_{\mathsf{let}}]{} T^{*\prime} \ .$$

Then $\|T^{*\prime}\| < \|T^*\|$.

Proof: Follows from Lemma A.8 and Lemma A.9. □

Lemma A.11. Let $T \in \Lambda_{\mathsf{let}}$, ι a decreasing weighting for T, and

$$(T, \iota) \xrightarrow[\mathsf{let}\text{-}\{\mathrm{V,C,A,GC}\}]{} (T', \iota') \ .$$

Then ι' is a decreasing weighting for T'.

Proof: As in [Bar81, Lemma 11.2.18(ii)]. □

Lemma A.12.

$$
\begin{tikzcd}
& M \arrow[dl, "\mathsf{let}\text{-I}"'] \arrow[dr, "\mathsf{let}\text{-I}"] & \\
M' \arrow[dr, dotted, "\mathsf{let}\text{-I}"'] & & M'' \arrow[dl, dotted, "\mathsf{let}\text{-I}"] \\
& \exists N &
\end{tikzcd}
$$

Proof: By a trivial structural induction. □

Lemma A.13, CR(let-I). let-I is confluent:

$$
\begin{tikzcd}
& M \arrow[dl, two heads, "\mathsf{let}\text{-I}"'] \arrow[dr, two heads, "\mathsf{let}\text{-I}"] & \\
M' \arrow[dr, dotted, two heads, "\mathsf{let}\text{-I}"'] & & M'' \arrow[dl, dotted, two heads, "\mathsf{let}\text{-I}"] \\
& \exists N &
\end{tikzcd}
$$

Proof: Follows trivially from Lemma A.12 by induction on the number of single steps from M to M' and from M to M'' as suggested by the following diagram: □

Lemma A.14, SN(let-I). let-I is strongly normalizing.

Proof: Follows from Lemma A.4 and Lemma A.8. □

Lemma A.15, SN(let-{V,C,A,GC}). let-{V,C,A,GC} is strongly normalizing.

Proof: Follows from Lemma A.4, Corollary A.10 and Lemma A.11. □

Lemma A.16, WCR(let-{V,C,A,GC}). let-{V,C,A,GC} is weakly Church-Rosser.

Proof: By a tedious but straightforward diagram chase. □

Lemma A.17, CR(let-{V,C,A,GC}). let-{V,C,A,GC} is confluent.

Proof: Follows from Lemma A.15 and Lemma A.16 by Newman's Lemma [Bar81, Proposition 3.1.25]. □

Lemma A.18. let-{V,C,A,GC} and let-I commute.

Proof: Again, by a tedious but straightforward diagram chase. □

Theorem A.19, CR(λ_{let}). λ_{let} is confluent:

$$
\begin{tikzcd}
& M \arrow[dl, two heads, "\mathsf{let}"] \arrow[dr, two heads, "\mathsf{let}"'] & \\
M' \arrow[dr, dotted, two heads, "\mathsf{let}"'] & & M'' \arrow[dl, dotted, two heads, "\mathsf{let}"] \\
& \exists N &
\end{tikzcd}
$$

Proof: Follows from Lemma A.13, Lemma A.17 and Lemma A.18 by the Lemma of Hindley-Rosen [Bar81, Proposition 3.3.5]. □

A.2 Standard Reduction

A.2.1 Preliminaries

Lemma A.20. Some simple observations:

a. For any M, $x \in \mathrm{fv}(M)$, there is no evaluation context E such that $E[x] \equiv \lambda y.M$.

b. Let C be a context, but not an evaluation context, and E be an evaluation context, $C[V] \equiv E[x]$. Then there exists an evaluation context E_0 and a non-evaluation context C_0 such that:

$$E_0[x] \equiv C[x] \qquad C[V] \equiv E[x]$$

$$E_0[V] \equiv C_0[x] \quad C_0[V] \equiv E[V] \ \cdot$$

Proof: (a.) Obvious. (b.) By a trivial structural induction on $C[V] \equiv E[x]$: there must be some point in the structure where the hole of each term is in different subterms (since the evaluation context will not "look inside" the value), and E_0 and C_0 can be contructed from those subterms. □

Lemma A.21. Let $A \in \mathcal{A}$. Then $A \not\to_s$.

Proof: By a simple structural induction on A.
Case 1: $A \equiv \lambda x.M$. Trivially, A is neither a redex nor formable into an evaluation context.
Case 2: $A \equiv \mathsf{let}\ x = M\ \mathsf{in}\ A'$. Follows from the inductive hypothesis and Lemma A.20.a.
□

Lemma A.22. Let $M \xrightarrow[\lambda_{\mathsf{let}} \backslash \lambda_{\mathsf{let}_s}]{} A$. Then $M \in \mathcal{A}$.

Proof: By structural induction over A.
Case 1: $A \equiv \lambda x.N$. Clearly A is not itself the contraction of any let-{I,V,C,A}-redex, so two cases are possible:
Case 1.A: $M \xrightarrow[\mathsf{let}\text{-GC}]{} A$. Then $M \equiv \mathsf{let}\ y = M'\ \mathsf{in}\ A \in \mathcal{A}$.
Case 1.B: $M \equiv \lambda x.M'$, $M' \xrightarrow[\lambda_{\mathsf{let}} \backslash \lambda_{\mathsf{let}_s}]{} N$. Clearly $M \in \mathcal{A}$.
Case 2: $A \equiv \mathsf{let}\ x = N\ \mathsf{in}\ A'$. By analysis of the position of the redex within M:
Case 2.A: $\langle M, A\rangle \in \lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}$. M cannot be a let-I-redex, as all top-level let-I-redexes are standard redexes. We also cannot have M a let-C-redex, which always produces an application, and hence never an answer.
Case 2.A.1: $\langle M, A\rangle \in \mathsf{let}\text{-V} \setminus \mathsf{let}_s\text{-V}$.
Case 2.A.2: $\langle M, A\rangle \in \mathsf{let}\text{-A} \setminus \mathsf{let}_s\text{-A}$.
Case 2.A.3: $\langle M, A\rangle \in \mathsf{let}\text{-GC} \setminus \mathsf{let}_s\text{-GC}$.
Case 2.B: So $M \equiv \mathsf{let}\ x = M'\ \mathsf{in}\ M''$, and one of the following:
Case 2.B.1: $M' \xrightarrow[\lambda_{\mathsf{let}} \backslash \lambda_{\mathsf{let}_s}]{} N$, $M'' \equiv A$, and so $M \in \mathcal{A}$.
Case 2.B.2: $M'' \xrightarrow[\lambda_{\mathsf{let}} \backslash \lambda_{\mathsf{let}_s}]{} A$. So by the induction hypothesis $M'' \in \mathcal{A}$, and then $M'' \in A$.
□

A.2.2 Reversing Non-Standard and Standard Sequences

The next four lemmas are dreadful, and correspond to what Barendregt achieves quite easily with a finiteness of developments theorem [Bar81, Th. 11.2.25, Lemmas 11.4.4 and 11.4.5]. This technique is not easily available to us since we have developed a bookkeeping technique only for one sort of redex; to keep track of all of the other, different redexes would be at least as messy as this approach.

Lemma A.23. Let $M \xrightarrow[\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}]{} N \xrightarrow[\mathsf{let}_s\text{-I}]{} N'$. Then there exists a term M' such that $M' \xrightarrow[\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}]{} N'$ and exactly one of the following is true:

1. $M \xrightarrow[\mathsf{let}_s\text{-I}]{} M'$.
2. There exists another term M'' such that $M \xrightarrow[\mathsf{let}_s\text{-C}]{} M'' \xrightarrow[\mathsf{let}_s\text{-I}]{} M'$.

Proof: By several case analyses:
<u>Case 1: $\langle M, N\rangle \in \lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}$</u>. We cannot have $\langle M, N\rangle \in \mathsf{let}\text{-I} \setminus \mathsf{let}_s\text{-I}$, since all top-level let-I-redexes are standard.
<u>Case 1.A</u>: $\langle M, N\rangle \in \mathsf{let}\text{-V} \setminus \mathsf{let}_s\text{-V}$. So

$$\begin{aligned} M &\equiv \mathsf{let}\ x = V\ \mathsf{in}\ C[x]\ , \\ N &\equiv \mathsf{let}\ x = V\ \mathsf{in}\ C[V] \equiv \mathsf{let}\ x = V\ \mathsf{in}\ E[(\lambda y.L_1)\, L_2]\ , \\ N' &\equiv \mathsf{let}\ x = V\ \mathsf{in}\ E[\mathsf{let}\ y = L_2\ \mathsf{in}\ L_1]\ , \end{aligned}$$

where C is not an evaluation context. We perform a structural induction on E:
Case 1.A.1: $E \equiv [\,]$. Trivial: take for instance $C \equiv (\lambda y.C_1[x])\, L_2$; then we have

$$\begin{aligned} M \equiv \mathsf{let}\ x = V\ \mathsf{in}\ C[x] \quad \xrightarrow[\mathsf{let}\text{-V}\setminus\mathsf{let}_s\text{-V}]{} \quad N &\equiv \mathsf{let}\ x = V\ \mathsf{in}\ C[V] \\ &\equiv \mathsf{let}\ x = V\ \mathsf{in}\ (\lambda y.L_1)\, L_2 \\ \xrightarrow[\mathsf{let}_s\text{-I}]{} \quad N' &\equiv \mathsf{let}\ x = V\ \mathsf{in}\ \mathsf{let}\ y = L_2\ \mathsf{in}\ L_1 \end{aligned}$$

So V must be a subterm of either L_1 or L_2; considering the case where $C \equiv (\lambda y.C_1)\, L_2$, C_1 again not an evaluation context, we have

$$\begin{aligned} M \equiv \mathsf{let}\ x = V\ \mathsf{in}\ (\lambda y.C_1[x])\, L_2 \quad &\xrightarrow[\mathsf{let}_s\text{-I}]{} \quad M' \equiv \mathsf{let}\ x = V\ \mathsf{in}\ \mathsf{let}\ y = L_2\ \mathsf{in}\ C_1[x] \\ &\xrightarrow[\mathsf{let}\text{-V}\setminus\mathsf{let}_s\text{-V}]{} \quad \mathsf{let}\ x = V\ \mathsf{in}\ \mathsf{let}\ y = L_2\ \mathsf{in}\ C_1[V] \ \equiv\ N' \end{aligned}$$

And similarly for V a subterm of L_1.
Case 1.A.2: $E \equiv E_1\, N_2$. So we have that C must also be an application; if we have $C \equiv E_1[(\lambda x.L_1)\, L_2]\, C_2$ then the result is trivial. Otherwise, for $C \equiv C_1\, N_2$ we must make an inductive argument; the key idea is that we must either have the expression filling the whole of C as a subterm of the standard contractum (as in Case 1.A.1) or in a different "branch" of the term (as in the previous alternative for this case).
Case 1.A.3: $E \equiv \mathsf{let}\ z = N_1\ \mathsf{in}\ E_1$, and
Case 1.A.4: $E \equiv \mathsf{let}\ z = E_2\ \mathsf{in}\ E_1[z]$. Similarly.
<u>Case 1.B</u>: $\langle M, N\rangle \in \mathsf{let}\text{-A} \setminus \mathsf{let}_s\text{-A}$. So

$$\begin{aligned} M &\equiv \mathsf{let}\ x = (\mathsf{let}\ y = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ M_2\ , \\ N &\equiv \mathsf{let}\ y = M_0\ \mathsf{in}\ \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2\ , \end{aligned}$$

where M is not a standard redex. We have that the standard redex contracts either M_0 to M_0', M_1 to M_1' or M_2 to M_2'. In the first case (respectively, the second and third) we have

$$N' \equiv \mathsf{let}\ y = M_0' \ \mathsf{in}\ \mathsf{let}\ x = M_1 \ \mathsf{in}\ M_2$$

($\mathsf{let}\ y = M_0 \ \mathsf{in}\ \mathsf{let}\ x = M_1' \ \mathsf{in}\ M_2$ and $\mathsf{let}\ y = M_0 \ \mathsf{in}\ \mathsf{let}\ x = M_1 \ \mathsf{in}\ M_2'$), and so

$$M' \equiv \mathsf{let}\ x = (\mathsf{let}\ y = M_0' \ \mathsf{in}\ M_1) \ \mathsf{in}\ M_2$$

($\mathsf{let}\ x = (\mathsf{let}\ y = M_0 \ \mathsf{in}\ M_1') \ \mathsf{in}\ M_2$ and $\mathsf{let}\ x = (\mathsf{let}\ y = M_0 \ \mathsf{in}\ M_1) \ \mathsf{in}\ M_2'$).
Case 1.C: $\langle M, N\rangle \in \mathsf{let}\text{-C} \setminus \mathsf{let}_s\text{-C}$. As in case 1.B.
Case 1.D: $\langle M, N\rangle \in \mathsf{let}\text{-GC}$. So $M \equiv \mathsf{let}\ x = M_0 \ \mathsf{in}\ N$, $x \notin \mathrm{fv}(N)$, and $M' \equiv \mathsf{let}\ x = M_0 \ \mathsf{in}\ N'$.
Case 2: $\langle M, N\rangle \notin \lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}$.
Case 2.A: $\langle N, N'\rangle \in \mathsf{let}_s\text{-I}$. So we have $N \equiv (\lambda x.N_1)\, N_2$, and we distinguish two subcases:
Case 2.A.1: $M \equiv (\mathsf{let}\ y = N_0 \ \mathsf{in}\ \lambda x.N_1)\, N_2$, $\langle \mathsf{let}\ y = N_0 \ \mathsf{in}\ \lambda x.N_1, \lambda x.N_1\rangle \in \mathsf{let}\text{-GC}$. $\lambda x.N_1 \in \mathcal{A}$, so we have

$$\begin{aligned}
&(\mathsf{let}\ y = N_0 \ \mathsf{in}\ \lambda x.N_1)\, N_2 \\
&\xrightarrow[\mathsf{let}_s\text{-C}]{} \ \mathsf{let}\ y = N_0 \ \mathsf{in}\ (\lambda x.N_1)\, N_2 \\
&\xrightarrow[\mathsf{let}_s\text{-I}]{} \ M' \equiv \mathsf{let}\ y = N_0 \ \mathsf{in}\ \mathsf{let}\ x = N_2 \ \mathsf{in}\ N_1 \\
&\xrightarrow[\mathsf{let}\text{-GC}]{} \ \mathsf{let}\ x = N_2 \ \mathsf{in}\ N_1 \equiv N' \ .
\end{aligned}$$

Case 2.A.2: Otherwise, the non-standard redex *must* be contained entirely within either N_1 or N_2, and we have the result by a simple diagram chase, with in either case $M' \equiv \mathsf{let}\ x = N_2^0 \ \mathsf{in}\ N_1^0$ where of the two pairs N_1^0 & N_1 and N_2^0 & N_2, in one pair the first reduces to the second, and in the other, the two are identical.
Case 2.B: $\langle N, N'\rangle \notin \mathsf{let}_s\text{-I}$. We have two possible cases for the structure of M:
Case 2.B.1: $M \equiv M_0\, M_3$. Then we have four simple subcases:
Case 2.B.1.a: $N' \equiv N_0'\, M_3$, $M_0 \xrightarrow[\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}]{} N_0 \xrightarrow[\mathsf{let}_s\text{-I}]{} N_0'$, and
Case 2.B.1.b: $N' \equiv M_0\, N_3'$, $M_3 \xrightarrow[\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}]{} N_3 \xrightarrow[\mathsf{let}_s\text{-I}]{} N_3'$. By the induction hypothesis.
Case 2.B.1.c: $N' \equiv M_0'\, M_3'$, $M_0 \xrightarrow[\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}]{} M_0'$, $M_3 \xrightarrow[\mathsf{let}_s\text{-I}]{} M_3'$, and
Case 2.B.1.d: $N' \equiv M_0'\, M_3'$, $M_0 \xrightarrow[\mathsf{let}_s\text{-I}]{} M_0'$, $M_3 \xrightarrow[\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}]{} M_3'$. Trivial.
Case 2.B.2: $M \equiv \mathsf{let}\ x = M_0 \ \mathsf{in}\ M_3$. Four similar subcases, each either by the induction hypothesis or trivial subterm analysis.

□

Lemma A.24.

$$\begin{tikzcd}
M \arrow[r, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] \arrow[d, dotted, "\mathsf{let}_s\text{-V}"] & N \arrow[d, "\mathsf{let}_s\text{-V}"] \\
M' \arrow[r, dotted, two heads, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] & N'
\end{tikzcd}$$

Proof: By several case analyses:
Case 1: $\langle M, N\rangle \in \lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}$. We again cannot have $\langle M, N\rangle \in \mathsf{let}\text{-I} \setminus \mathsf{let}_s\text{-I}$, since all top-level let-I-redexes are standard.

Case 1.A: $\langle M, N\rangle \in \mathsf{let}\text{-V} \setminus \mathsf{let}_s\text{-V}$. So

$$\begin{aligned} M &\equiv \mathsf{let}\ x = V\ \mathsf{in}\ C[x]\ , \\ N &\equiv \mathsf{let}\ x = V\ \mathsf{in}\ C[V]\ , \end{aligned}$$

where C is not an evaluation context. As in case 1.A, Lemma A.23, except that more than one non-standard contraction may be required if the original non-standard redex is within the value which the standard redex dereferences.

Case 1.B: $\langle M, N\rangle \in \mathsf{let}\text{-A} \setminus \mathsf{let}_s\text{-A}$. So

$$\begin{aligned} M &\equiv \mathsf{let}\ x = (\mathsf{let}\ y = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ M_2\ , \\ N &\equiv \mathsf{let}\ y = M_0\ \mathsf{in}\ \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2\ , \end{aligned}$$

where M is not a standard redex. We have three cases, revolving around this constraint on M:

Case 1.B.1: $M_1 \in \mathcal{A}$, $(\nexists E)\, M_2 \equiv E[x]$. We must have the standard redex is internal to M_2, and so

$$M' \equiv \mathsf{let}\ x = (\mathsf{let}\ y = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ M_2'\ .$$

Case 1.B.2: $M_2 \equiv E[x]$, $M_1 \notin \mathcal{A}$. Then the standard redex is either internal to M_1, or else internal to M_0 with $M_1 \equiv E_0[y]$; in the former case we have

$$M' \equiv \mathsf{let}\ x = (\mathsf{let}\ y = M_0\ \mathsf{in}\ M_1')\ \mathsf{in}\ M_2\ ,$$

and in the latter case,

$$M' \equiv \mathsf{let}\ x = (\mathsf{let}\ y = M_0'\ \mathsf{in}\ M_1)\ \mathsf{in}\ M_2\ .$$

Case 1.B.3: $M_1 \notin \mathcal{A}$, $(\nexists E)\, M_2 \equiv E[x]$. As in Case 1.B.1 above.

Case 1.C: $\langle M, N\rangle \in \mathsf{let}\text{-C} \setminus \mathsf{let}_s\text{-C}$. So we have $M \equiv (\mathsf{let}\ x = M_0\ \mathsf{in}\ M_1)\, M_2$, $M_1 \notin \mathcal{A}$; clearly the standard redex is internal to M_1, $M_1 \xrightarrow[\mathsf{let}_s\text{-V}]{} M_1'$, and we have

$$M' \equiv (\mathsf{let}\ x = M_0\ \mathsf{in}\ M_1')\, M_2\ .$$

Case 1.D: $\langle M, N\rangle \in \mathsf{let}\text{-GC}$. So $M \equiv \mathsf{let}\ x = M_0\ \mathsf{in}\ N$, $x \notin \mathrm{fv}(N)$, and $M' \equiv \mathsf{let}\ x = M_0\ \mathsf{in}\ N'$.

Case 2: $\langle M, N\rangle \notin \lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}$.

Case 2.A: $\langle N, N'\rangle \in \mathsf{let}_s\text{-V}$. So $N \equiv \mathsf{let}\ x = V\ \mathsf{in}\ E[x]$. Where the non-standard reduction is a subterm of V, we have the result trivially; where it is a subterm of $E[x]$, the result follows from Lemma A.20.b.

Case 2.B: $\langle N, N'\rangle \notin \mathsf{let}_s\text{-V}$. As in Case 2.B of Lemma A.23.

□

Lemma A.25.

$$\begin{tikzcd} M \arrow[r, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] \arrow[d, dotted, ->>, "\mathsf{let}_s\text{-A}"] & N \arrow[d, "\mathsf{let}_s\text{-A}"] \\ M' \arrow[r, dotted, ->>, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] & N' \end{tikzcd}$$

Proof: By several case analyses:
Case 1: $\langle M, N\rangle \in \lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}$. We again cannot have $\langle M, N\rangle \in \mathsf{let}$-I $\setminus$ let_s-I, since all top-level let-I-redexes are standard.
Case 1.A: $\langle M, N\rangle \in \mathsf{let}$-V $\setminus$ let_s-V. So

$$\begin{aligned} M &\equiv \mathsf{let}\ x = V\ \mathsf{in}\ C[x]\ , \\ N &\equiv \mathsf{let}\ x = V\ \mathsf{in}\ C[V]\ , \end{aligned}$$

where C is not an evaluation context. As in case 1.A, Lemma A.23.
Case 1.B: $\langle M, N\rangle \in \mathsf{let}$-A $\setminus$ let_s-A. So

$$\begin{aligned} M &\equiv \mathsf{let}\ x = (\mathsf{let}\ y = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ M_2\ , \\ N &\equiv \mathsf{let}\ y = M_0\ \mathsf{in}\ \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2\ , \end{aligned}$$

where M is not a standard redex. We again have three cases about the non-standardness of M's contraction.
Case 1.B.1: $M_1 \in \mathcal{A}$, $(\nexists E)\, M_2 \equiv E[x]$. As in Case 1.B.1 of Lemma A.24.
Case 1.B.2: $M_2 \equiv E_0[x]$, $M_1 \notin \mathcal{A}$.
Case 1.B.2.a: $M_1 \equiv E_1[y]$.
Case 1.B.2.a.i: $M_0 \in \mathcal{V}_s$. So we have

$$\begin{aligned} M &\equiv \mathsf{let}\ x = \mathsf{let}\ y = \lambda z.M_3 \\ &\qquad\qquad\qquad \mathsf{in}\ E_1[y] \\ &\qquad\ \ \mathsf{in}\ E_0[x]\ , \text{ and} \\ M \xrightarrow[\mathsf{let}_s\text{-V}]{} M'' &\equiv \mathsf{let}\ x = \mathsf{let}\ y = \lambda z.M_3 \\ &\qquad\qquad\qquad \mathsf{in}\ E_1[\lambda z.M_3] \\ &\qquad\ \ \mathsf{in}\ E_0[x]\ . \end{aligned}$$

If $E_1[\lambda z.M_3]$ is then an answer, then we have $M'' \xrightarrow[\mathsf{let}_s\text{-A}]{} M' \equiv N'$, otherwise $M'' \equiv M'$.
Case 1.B.2.a.ii: $M_0 \in \mathcal{A} \setminus \mathcal{V}_s$. So we have

$$\begin{aligned} M &\equiv \mathsf{let}\ x = \mathsf{let}\ y = (\mathsf{let}\ z = M_3\ \mathsf{in}\ A_0) \\ &\qquad\qquad\qquad \mathsf{in}\ E_1[y] \\ &\qquad\ \ \mathsf{in}\ E_0[x]\ , \text{ and} \\ M \xrightarrow[\mathsf{let}_s\text{-A}]{} M' &\equiv \mathsf{let}\ x = \mathsf{let}\ z = M_3 \\ &\qquad\qquad\qquad \mathsf{in}\ \mathsf{let}\ y = A_0 \\ &\qquad\qquad\qquad\quad \mathsf{in}\ E_1[y] \\ &\qquad\ \ \mathsf{in}\ E_0[x]\ . \end{aligned}$$

and a particularly nasty diagram chase,

$$\begin{tikzcd}[column sep=small]
M \equiv \begin{array}{l}\mathsf{let}\ x = \mathsf{let}\ y = \mathsf{let}\ z = M_3 \\ \qquad\qquad\qquad \mathsf{in}\ A_0 \\ \qquad\quad \mathsf{in}\ E_1[y] \\ \quad \mathsf{in}\ E_0[x]\end{array} \arrow[rr, "\mathsf{let}\text{-A} \setminus \mathsf{let}_s\text{-A}"'] \arrow[d, "\mathsf{let}_s\text{-A}"] & & N \equiv \begin{array}{l}\mathsf{let}\ y = \mathsf{let}\ z = M_3 \\ \qquad\qquad \mathsf{in}\ A_0 \\ \quad \mathsf{in}\ \mathsf{let}\ x = E_1[y] \\ \quad\quad \mathsf{in}\ E_0[x]\end{array} \arrow[d, "\mathsf{let}_s\text{-A}"] \\
M' \equiv \begin{array}{l}\mathsf{let}\ x = \mathsf{let}\ z = M_3 \\ \qquad\quad \mathsf{in}\ \mathsf{let}\ y = A_0 \\ \qquad\qquad \mathsf{in}\ E_1[y] \\ \quad \mathsf{in}\ E_0[x]\ .\end{array} \arrow[r, "\mathsf{let}\text{-A} \setminus \mathsf{let}_s\text{-A}"'] & N'' \equiv \begin{array}{l}\mathsf{let}\ z = M_3 \\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{let}\ y = A_0 \\ \qquad\qquad\quad \mathsf{in}\ E_1[y] \\ \quad \mathsf{in}\ E_0[x]\end{array} \arrow[r, "\mathsf{let}\text{-A} \setminus \mathsf{let}_s\text{-A}"'] & N' \equiv \begin{array}{l}\mathsf{let}\ z = M_3 \\ \mathsf{in}\ \mathsf{let}\ y = A_0 \\ \quad \mathsf{in}\ \mathsf{let}\ x = E_1[y] \\ \quad\quad \mathsf{in}\ E_0[x]\end{array}
\end{tikzcd}$$

Case 1.B.2.a.iii: $M_0 \notin \mathcal{A}$. Then the standard contraction is clearly internal to M_0, and M' is trivially constructed.
Case 1.B.2.b: $(\nexists E_1)\, M_1 \equiv E_1[y]$. Trivially, the standard redex is a subterm of M_0, and we have the result by a simple diagram chase.
Case 1.B.3: $M_1 \notin \mathcal{A}$, $(\nexists E)\, M_2 \equiv E[x]$. As in Case 1.B.1 of Lemma A.24.
Case 1.C: $\langle M, N\rangle \in \mathsf{let}\text{-C} \setminus \mathsf{let}_s\text{-C}$. As in Case 1.C of Lemma A.24.
Case 1.D: $\langle M, N\rangle \in \mathsf{let}\text{-GC}$. So $M \equiv \mathsf{let}\ x = M_0\ \mathsf{in}\ N$, $x \notin \mathrm{fv}(N)$, and $M' \equiv \mathsf{let}\ x = M_0\ \mathsf{in}\ N'$.
Case 2: $\langle M, N\rangle \notin \lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}$.
Case 2.A: $\langle N, N'\rangle \in \mathsf{let}_s\text{-A}$. So $N \equiv \mathsf{let}\ x = \mathsf{let}\ y = N_0\ \mathsf{in}\ A\ \mathsf{in}\ E[x]$, and we distinguish two cases:
Case 2.A.1: $M \equiv \mathsf{let}\ x = (\mathsf{let}\ z = (\mathsf{let}\ y = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ A)\ \mathsf{in}\ E[x]$, and

$$\langle \mathsf{let}\ z = (\mathsf{let}\ y = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ A, \mathsf{let}\ y = N_0\ \mathsf{in}\ A\rangle \in \mathsf{let}\text{-A} \setminus \mathsf{let}_s\text{-A}\ \ .$$

Case 2.A.2: Otherwise, the non-standard redex is internal, and produces N_0, A or $E[x]$, and in either case we have the result by a simple diagram chase.
Case 2.B: $\langle N, N'\rangle \notin \mathsf{let}_s\text{-A}$. As in Case 2.B of Lemma A.23.
□

Corollary A.26.

$$\begin{tikzcd}
M \arrow[r, two heads, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] \arrow[d, dotted, "\mathsf{let}_s\text{-}\{\mathrm{V,A}\}"] & N \arrow[d, "\mathsf{let}_s\text{-}\{\mathrm{V,A}\}"] \\
M' \arrow[r, dotted, two heads, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] & N'
\end{tikzcd}$$

Proof: By induction over the length of the reduction sequence from M to N. □

Lemma A.27. Let $M \xrightarrow[\lambda_{\mathsf{let}}\setminus\lambda_{\mathsf{let}_s}]{} N \xrightarrow[\mathsf{let}_s\text{-C}]{} N'$. Then one of the following must be true:

1. There exists some term M' such that

$$M \xrightarrow[\mathsf{let}_s\text{-C}]{} M' \xrightarrow[\lambda_{\mathsf{let}}\setminus\lambda_{\mathsf{let}_s}]{} N'\ \ .$$

2. There exists terms M' and N'' such that both

$$M \xrightarrow[\mathsf{let}_s\text{-C}]{} M' \xrightarrow[\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}]{} N''$$

and

$$N' \xrightarrow[\mathsf{let}_s\text{-C}]{} N'' \ .$$

Proof: By several case analyses:

Case 1: $\langle M, N\rangle \in \lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}$. The subcases are as in the previous proofs:

Case 1.A: $\langle M, N\rangle \in \mathsf{let}\text{-V} \setminus \mathsf{let}_s\text{-V}$. So

$$\begin{aligned} M &\equiv \mathsf{let}\ x = V\ \mathsf{in}\ C[x] \ , \\ N &\equiv \mathsf{let}\ x = V\ \mathsf{in}\ C[V] \ , \end{aligned}$$

where C is not an evaluation context. As in case 1.A, Lemma A.23, with $N' \equiv N''$.

Case 1.B: $\langle M, N\rangle \in \mathsf{let}\text{-A} \setminus \mathsf{let}_s\text{-A}$. So

$$\begin{aligned} M &\equiv \mathsf{let}\ x = (\mathsf{let}\ y = M_0\ \mathsf{in}\ M_1)\ \mathsf{in}\ M_2 \ , \\ N &\equiv \mathsf{let}\ y = M_0\ \mathsf{in}\ \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 \ , \end{aligned}$$

where M is not a standard redex. We again have three cases about the non-standardness of M's contraction. As in Case 1.B of Lemma A.24, with $N' \equiv N''$.

Case 1.C: $\langle M, N\rangle \in \mathsf{let}\text{-C} \setminus \mathsf{let}_s\text{-C}$. So

$$\begin{aligned} M &\equiv (\mathsf{let}\ x = M_0\ \mathsf{in}\ M_1)\, M_2,\ M_1 \notin \mathcal{A} \\ N &\equiv \mathsf{let}\ x = M_0\ \mathsf{in}\ M_1\, M_2 \ . \end{aligned}$$

Note that we cannot have $\langle M_1\, M_2, N_0\rangle \in \mathsf{let}_s\text{-C}$, since $M_1 \notin \mathcal{A}$. So the standard redex must be within M_1, $M_1 \xrightarrow[\mathsf{let}_s\text{-C}]{} N_1$, $M'' \equiv (\mathsf{let}\ x = M_0\ \mathsf{in}\ N_1)\, M_2$. If we have $N_1 \in \mathcal{A}$, then $M'' \xrightarrow[\mathsf{let}_s\text{-C}]{} M' \equiv N'$; otherwise $M'' \equiv M' \xrightarrow[\mathsf{let}\text{-C} \setminus \mathsf{let}_s\text{-C}]{} N' \equiv N''$.

Case 1.D: $\langle M, N\rangle \in \mathsf{let}\text{-GC}$. So $M \equiv \mathsf{let}\ x = M_0\ \mathsf{in}\ N$, $x \notin \mathrm{fv}(N)$, and $M' \equiv \mathsf{let}\ x = M_0\ \mathsf{in}\ N'$, $N' \equiv N''$.

Case 2: $\langle M, N\rangle \notin \lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}$.

Case 2.A: $\langle N, N'\rangle \in \mathsf{let}_s\text{-C}$. So $N \equiv N_1\, N_2$, $N_1 \equiv \mathsf{let}\ x = N_3\ \mathsf{in}\ A$ and $N' \equiv \mathsf{let}\ x = N_3\ \mathsf{in}\ A\, N_2$.

Case 2.A.1: $M \equiv N_1\, N_0$, $N_0 \xrightarrow[\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}]{} N_2$. Trivial; we have $M' \equiv \mathsf{let}\ x = N_3\ \mathsf{in}\ A\, N_0$ and $N' \equiv N''$.

Case 2.A.2: $M \equiv N_4\, N_2$, $N_4 \xrightarrow[\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}]{} N_1$. To this point we have

$$\begin{tikzcd}
N_4\, N_2 \equiv M \arrow[r, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] & N \equiv N_1\, N_2 \arrow[d, leftrightarrow, "\equiv"] \\
& (\mathsf{let}\ x = N_3\ \mathsf{in}\ A)\, N_2 \arrow[d, "\mathsf{let}_s\text{-C}"] \\
& N' \equiv \mathsf{let}\ x = N_3\ \mathsf{in}\ A\, N_2
\end{tikzcd}$$

We have two cases depending on whether N_4 is itself the contractum.
Case 2.A.2.a: $\langle N_4, N_1\rangle \in \lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}$. By analysis of the specific form of reduction. Note that we cannot have $\langle N_4, N_1\rangle \in \mathsf{let}\text{-I} \setminus \mathsf{let}_s\text{-I}$; such would be a standard redex. We also cannot have $\langle N_4, N_1\rangle \in \mathsf{let}\text{-C} \setminus \mathsf{let}_s\text{-C}$ since N_1 would not be an answer, and hence $\langle N, N'\rangle$ not a standard redex.
Case 2.A.2.a.i: $\langle N_4, N_1\rangle \in \mathsf{let}\text{-V} \setminus \mathsf{let}_s\text{-V}$. So $N_4 \equiv \mathsf{let}\ x = N_3\ \mathsf{in}\ A_0$ (we have $A_0 \in \mathcal{A}$ by Lemma A.22) and $A_0 \xrightarrow[\mathsf{let}\text{-V}\setminus\mathsf{let}_s\text{-V}]{} A$; then:

$$M \equiv N_4\ N_2 \xrightarrow[\mathsf{let}_s\text{-C}]{} M' \equiv \mathsf{let}\ x = N_3\ \mathsf{in}\ A_0\ N_2 \xrightarrow[\mathsf{let}\text{-V}\setminus\mathsf{let}_s\text{-V}]{} N'' \equiv N' \ .$$

Case 2.A.2.a.ii: $\langle N_4, N_1\rangle \in \mathsf{let}\text{-A} \setminus \mathsf{let}_s\text{-A}$. So we have

$$\begin{array}{rcl} A & \equiv & \mathsf{let}\ y = M_6 \\ & & \ \mathsf{in}\ A_0 \\ N_4 & \equiv & \mathsf{let}\ y = \mathsf{let}\ x = N_3\ \mathsf{in}\ M_6 \\ & & \ \mathsf{in}\ A_0 \end{array} ,$$

and then

$$\begin{tikzcd}
\begin{array}{l} M \equiv (\mathsf{let}\ y = \mathsf{let}\ x = N_3\ \mathsf{in}\ M_6 \\ \qquad \mathsf{in}\ A_0)\ N_2 \end{array} \arrow[r, "\mathsf{let}\text{-A} \setminus \mathsf{let}_s\text{-A}"'] \arrow[dd, "\mathsf{let}_s\text{-C}"] & \begin{array}{l} N \equiv (\mathsf{let}\ x = N_3 \\ \qquad \mathsf{in}\ \mathsf{let}\ y = M_6 \\ \qquad\quad \mathsf{in}\ A_0)\ N_2 \end{array} \arrow[d, "\mathsf{let}_s\text{-C}"] \\
 & \begin{array}{l} N' \equiv \mathsf{let}\ x = N_3 \\ \qquad \mathsf{in}\ (\mathsf{let}\ y = M_6 \\ \qquad\quad \mathsf{in}\ A_0)\ N_2 \end{array} \arrow[d, "\mathsf{let}_s\text{-C}"] \\
\begin{array}{l} M' \equiv \mathsf{let}\ y = \mathsf{let}\ x = N_3\ \mathsf{in}\ M_6 \\ \qquad \mathsf{in}\ A_0\ N_2 \end{array} \arrow[r, "\mathsf{let}\text{-A} \setminus \mathsf{let}_s\text{-A}"'] & \begin{array}{l} N'' \equiv \mathsf{let}\ x = N_3 \\ \qquad \mathsf{in}\ \mathsf{let}\ y = M_6 \\ \qquad\quad \mathsf{in}\ A_0\ N_2 \end{array}
\end{tikzcd}$$

Case 2.A.2.a.iii: $\langle N_4, N_1\rangle \in \mathsf{let}\text{-GC}$. So $N_4 \equiv \mathsf{let}\ y = M_7\ \mathsf{in}\ N_1$. Expanding terms, we have:

$$\begin{tikzcd}
\begin{array}{l} M \equiv (\mathsf{let}\ y = M_7 \\ \qquad \mathsf{in}\ \mathsf{let}\ x = N_3 \\ \qquad\quad \mathsf{in}\ A)\ N_2 \end{array} \arrow[r, "\mathsf{let}\text{-GC}"'] \arrow[d, "\mathsf{let}_s\text{-C}"] & \begin{array}{l} N \equiv (\mathsf{let}\ x = N_3 \\ \qquad \mathsf{in}\ A)\ N_2 \end{array} \arrow[dd, "\mathsf{let}_s\text{-C}"] \\
\begin{array}{l} M'' \equiv \mathsf{let}\ y = M_7 \\ \qquad \mathsf{in}\ (\mathsf{let}\ x = N_3 \\ \qquad\quad \mathsf{in}\ A)\ N_2 \end{array} \arrow[d, "\mathsf{let}_s\text{-C}"] & \\
\begin{array}{l} M' \equiv \mathsf{let}\ y = M_7 \\ \qquad \mathsf{in}\ \mathsf{let}\ x = N_3 \\ \qquad\quad \mathsf{in}\ A\ N_2 \end{array} \arrow[r, "\mathsf{let}\text{-GC}"'] & \begin{array}{l} N'' \equiv N' \equiv \mathsf{let}\ x = N_3 \\ \qquad \mathsf{in}\ A\ N_2 \end{array}
\end{tikzcd}$$

Case 2.A.2.b: $\langle N_4, N_1 \rangle \notin \lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}$. So $N_4 \equiv \mathsf{let}\ x = N_5\ \mathsf{in}\ N_6$ where one of the following is true:

Case 2.A.2.b.i: $N_5 \xrightarrow[\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}]{} N_3$ and $N_6 \equiv A$. Trivially, $M' \equiv \mathsf{let}\ x = N_5\ \mathsf{in}\ (A\ N_2)$, $N' \equiv N''$.

Case 2.A.2.b.ii: $N_6 \xrightarrow[\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}]{} A$ and $N_3 \equiv N_5$. Then by Lemma A.22, $N_6 \in \mathcal{A}$, and we have $M' \equiv \mathsf{let}\ x = N_3\ \mathsf{in}\ (N_6\ N_2)$, $N' \equiv N''$.

<u>Case 2.B</u>: $\langle N, N' \rangle \notin \mathsf{let}_s$-C. We have $N' \equiv N''$, with two possible cases for the structure of M as in Case 2.B of Lemma A.23.

□

Corollary A.28.

$$
\begin{tikzcd}
M \arrow[rr, two heads, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] \arrow[d, dotted, two heads, "\mathsf{let}_s\text{-C}"] & & N \arrow[d, "\mathsf{let}_s\text{-C}"] \\
M' \arrow[drr, dotted, two heads, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] & & N' \arrow[d, dotted, two heads, "\mathsf{let}_s\text{-C}"] \\
& & N''
\end{tikzcd}
$$

Proof: By induction on the length of the reduction sequence from M to N. □

Corollary A.29.

$$
\begin{tikzcd}
M \arrow[r, two heads, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] \arrow[d, dotted, two heads, "\mathsf{let}_s\text{-C}"] & N \arrow[dd, "\mathsf{let}_s\text{-I}"] \\
M_0 \arrow[d, dotted, two heads, "\mathsf{let}_s\text{-I}"] & \\
M' \arrow[r, dotted, two heads, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] & N'
\end{tikzcd}
,
$$

with $M_0 \not\equiv M'$.

Proof: Since the sequence of let_s-C-reductions is necessarily followed by at least one let_s-I-contraction, we have an automatic upper limit on the length of the $N' \to N''$ sequence which Corollary A.28 does not explicity limit. □

A.2.3 Main Standardization Lemmas

Lemma A.30. For $N' \not\xrightarrow[\lambda_{\mathsf{let}_s}]{}$,

$$
\begin{tikzcd}
M \arrow[r, two heads, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] \arrow[d, dotted, two heads, "\lambda_{\mathsf{let}_s}"] & N \arrow[d, "\lambda_{\mathsf{let}_s}"] \\
M' \arrow[r, dotted, two heads, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] & N' \not\xrightarrow[\lambda_{\mathsf{let}_s}]{}
\end{tikzcd}
$$

Proof: Follows from Corollary A.26, Corollary A.28 and Corollary A.29. We need not worry about a trailing let_s-C-sequence since we know N' to be in λ_{let_s}-normal form. □

Lemma A.31. For $N' \not\xrightarrow[\lambda_{\mathsf{let}_s}]{}$,

$$\begin{tikzcd}
M \arrow[r, ->>, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] \arrow[d, dotted, ->>, "\lambda_{\mathsf{let}_s}"] & N \arrow[d, ->>, "\lambda_{\mathsf{let}_s}"] \\
M' \arrow[r, dotted, ->>, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"'] & N' \not\xrightarrow[\lambda_{\mathsf{let}_s}]{}
\end{tikzcd}$$

Proof: By induction on the length of the reduction sequence from N to N'. The base case is Lemma A.30; the inductive case has

$$N \xrightarrow[\lambda_{\mathsf{let}_s}]{} N_0 \xrightarrow[\lambda_{\mathsf{let}_s}]{} \!\!\!\!\rightarrow N'$$

then based on the specific sort of contraction from N to N_0 we use either Corollary A.29, Corollary A.26 or Corollary A.28. The third case requires a partition of the $N_0 \twoheadrightarrow_{\lambda_{\mathsf{let}_s}} N'$ sequence into $N_0 \twoheadrightarrow_{\lambda_{\mathsf{let}_s}} N'_0 \twoheadrightarrow_{\lambda_{\mathsf{let}_s}} N'$: The first half of the partition is that required as the trailing let_s-C-sequence, and is thus resolved; the inductive step is on $N'_0 \twoheadrightarrow_{\lambda_{\mathsf{let}_s}} N'$ only. □

Lemma A.32. For $N \not\xrightarrow[\lambda_{\mathsf{let}_s}]{}$,

$$\begin{tikzcd}
M \arrow[dd, ->>, "\lambda_{\mathsf{let}}"'] \arrow[dr, dotted, ->>, "\lambda_{\mathsf{let}_s}"] & \\
& \exists M' \arrow[dl, dotted, ->>, "\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}"] \\
N \not\xrightarrow[\lambda_{\mathsf{let}_s}]{} &
\end{tikzcd} .$$

Proof: Any reduction $M \twoheadrightarrow_{\lambda_{\mathsf{let}}} N$ can be written as

$$\begin{aligned}
M &\twoheadrightarrow_{\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}} M_{1,0} \twoheadrightarrow_{\lambda_{\mathsf{let}_s}} M_{1,1} \\
&\twoheadrightarrow_{\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}} M_{2,0} \twoheadrightarrow_{\lambda_{\mathsf{let}_s}} M_{2,1} \\
&\qquad\qquad \vdots \\
&\twoheadrightarrow_{\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}} M_{n,0} \twoheadrightarrow_{\lambda_{\mathsf{let}_s}} M_{n,1} \equiv N \ .
\end{aligned}$$

So the result follows from Lemma A.31 by induction on n. □

Theorem A.33, SR(λ_{let}). λ_{let_s} is a standard reduction relation for λ_{let}:

$$M \twoheadrightarrow_{\lambda_{\mathsf{let}}} A \Longrightarrow (\exists A')\; M \twoheadrightarrow_{\lambda_{\mathsf{let}_s}} A' \ .$$

Proof: By Lemma A.21 we know that A is a λ_{let_s}-normal form. So by Lemma A.32 we have M',

$$M \twoheadrightarrow_{\lambda_{\mathsf{let}_s}} M' \twoheadrightarrow_{\lambda_{\mathsf{let}} \setminus \lambda_{\mathsf{let}_s}} A \ .$$

Then by Lemma A.22, $M' \in \mathcal{A}$. □

A.3 Uniform Monotonicity of Standard Call-by-Name Reduction

This appendix proves a property of call-by-name λ-calculus, namely that β-reduction does not increase the length of a standard reduction sequence from a term to an answer.

Notation Let $\underset{s}{\to}$ be head reduction in call-by-name λ-calculus and let $\underset{i}{\to}$ denote reduction of a non-head redex, i.e. $\underset{i}{\to} \cup \underset{s}{\to} = \underset{\beta}{\to}$. Furthermore, $M \underset{i,1}{\longrightarrow} N$ means that we have not only $\sigma : M \underset{i}{\to} N$, but also some collection of redexes $\mathcal{F}$ contained in M such that $\sigma : (M, \mathcal{F}) \twoheadrightarrow^{\mathrm{cpl}} N$. If $\to$ is some reduction relation, we write $M \to^n N$ to express that M reduces in at most n steps to N.

Lemma A.34.

$$\begin{tikzcd}
M \arrow[r, "s"'] \arrow[d, "{i,1}"] & M' \arrow[d, dotted, "i"'] \\
N \arrow[r, dotted, "s"'] & \exists N'
\end{tikzcd} .$$

Proof: Take $\sigma : M \xrightarrow{\Delta}_s M'$ and $\iota : (M, \mathcal{F}) \twoheadrightarrow_i^{\mathrm{cpl}} N$. By Lemma 11.4.3(ii) of Barendregt, $\frac{\Delta}{\iota} = \{\Delta'\}$, where Δ' is the head redex of N; take N_1 such that $N \xrightarrow{\Delta'}_s N_1$. Also, by Lemma 11.4.3(iii) of Barendregt, every $\Delta_i \in \frac{\mathcal{F}}{\sigma}$ is an internal redex of N; take N_2 such that $(M', \frac{\mathcal{F}}{\sigma}) \twoheadrightarrow_i^{\mathrm{cpl}} N_2$. Now we have that both

$$\iota' : M \overset{\mathcal{F}}{\twoheadrightarrow}_{1,i} N \xrightarrow{\frac{\Delta}{\iota}} N_1$$

and

$$\sigma' : M \xrightarrow{\Delta}_s M' \overset{\frac{\mathcal{F}}{\sigma}}{\twoheadrightarrow} N_2$$

are complete developments of $(M, \mathcal{F} \cup \{\Delta\})$, so by FD $N' \equiv N_1 \equiv N_2$. $\square$

Corollary A.35.

$$\begin{tikzcd}
M \arrow[r, "s"'] \arrow[d, "i"] & M' \arrow[d, dotted, "i"'] \\
N \arrow[r, dotted, "s"'] & \exists N'
\end{tikzcd} .$$

Proof: Follows from Lemma A.34 since

$$M \xrightarrow{\Delta}_i N \implies (M, \{\Delta\}) \twoheadrightarrow_i^{\mathrm{cpl}} N .$$

$\square$

Proposition 5.7 Let $M, N \in \Lambda$, such that $\lambda \vdash M \twoheadrightarrow N$ and $\lambda \vdash M \underset{s}{\rightarrow}^n A$, for some answer A, $n \geq 0$. Then $N \underset{s}{\rightarrow}^n A'$ for some answer A'.

Proof: We use an induction on n. If $n = 0$, we have $M \equiv A \twoheadrightarrow N$. By the definition of answers A is an abstraction. Hence N is also an abstraction and therefore an answer.

For the inductive step assume $M \underset{s}{\rightarrow} M' \underset{s}{\rightarrow}^{n-1} A$ and $M \twoheadrightarrow N$. We use an induction on the length m of reduction from M to N.

If $m = 0$, the proposition follows immediately. Otherwise let N' be such that $M \rightarrow N' \rightarrow^{m-1} N$. If $M \underset{s}{\rightarrow} M'$, it follows that $N' \equiv M'$ since standard reduction is deterministic. By the outer induction hypothesis, $N' \underset{s}{\rightarrow}^{n-1} A'$, for some answer A'. Then, by the inner induction hypothesis, $N \underset{s}{\rightarrow}^{n-1} A'$.

On the other hand, if not $M \underset{s}{\rightarrow} N'$, it must hold that $M \underset{i}{\rightarrow} N'$. By Corollary A.35, there is an N'' such that $N' \underset{s}{\rightarrow} N''$ and $M' \underset{i}{\twoheadrightarrow} N''$. By the outer induction hypothesis there is an answer A' such that $N'' \underset{s}{\rightarrow}^{n-1} A'$. Hence, $N' \underset{s}{\rightarrow}^{n} A'$, and, by the inner induction hypothesis, $N \underset{s}{\rightarrow}^{n} A'$. $\square$