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

John Marais	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

---

†Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, Postfach 6980, 76128 Karlsruhe, Germany. Email, {marais,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

\[
\begin{align*}
\text{let } x &= 1 + 2 \text{ in} \\
\text{let } f &= \lambda y.x + y \text{ in} \\
f &\ y + f \ y
\end{align*}
\]

the variable \( x \) appears textually only once but substituting \( 1 + 2 \) for \( x \) in the body of the \textit{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 \textit{let} construct. The reduction rules appear more intuitive if a \textit{let} construct is used but an equivalent calculus can be formed without \textit{let} using the usual equivalence between \textit{let} \( x = M \) 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 $\rightarrow$ to be the compatible closure of $\beta$ and $\rightarrow^*$ to be the reflexive-transitive closure of $\rightarrow$ omitting subscripts when possible without confusion. We write $M \rightarrow N$ to mean that we have $M \equiv E[\Delta] \Gamma N \equiv E[\Delta']$ and $\langle \Delta, \Delta' \rangle \in \beta \Gamma$ with $\rightarrow^*$ as the reflexive-transitive closure of $\rightarrow$.

Throughout this report we use the following notational conventions. We use $\text{fv}(M)$ 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 $\alpha$-renaming) and reserve $M = N$ for convertibility. Following Barendregt [Bar81], we work with equivalence classes of $\alpha$-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

<table>
<thead>
<tr>
<th></th>
<th>x, y, z</th>
</tr>
</thead>
</table>

Values 

V, W ::={ x | λx. M }

Terms 

L, M, N ::={ V | M N | let x = M in N }

Evaluation Contexts 

E ::={ [] | E M }

Reduction Rule

(β) (λx.M) N M[x := N]

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

Syntactic Domains

<table>
<thead>
<tr>
<th></th>
<th>x, y, z</th>
</tr>
</thead>
</table>

Values 

V, W ::={ x | λx. M }

Terms 

L, M, N ::={ V | M N | let x = M in N }

Reduction Rules

(let-I) (λx.M) N → let x = N in M

(let-V) let x = V in C[x] → let x = V in C[V]

(let-C) (let x = L in M) N → let x = L in M N

(let-A) let y = (let x = L in M) in N → let x = L in let y = M in N

(let-GC) let x = M in N → N if x /∈ fv N

Figure 2: The call-by-need λ-calculus λlet.

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 εR to express that M reduces in one R-reduction step to N and M εR 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 λlet. 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 identifier. Hence sharing in a function graph corresponds to naming in a term.

The second half of Figure 2 presents reduction rules for λlet:

- Rule let-I introduces a let binding from an application. Given an application (λx.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 $N$. \texttt{let-I} models this by representing the reference with a \texttt{let-bound name}.

- Dereferencing is expressed by rule \texttt{let-VI} which substitutes a defining value for a variable occurrence. Note that $\Gamma$ 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 $\Gamma$ shared expression.

- Rule \texttt{let-C} allows \texttt{let}-bindings to commute with applications $\Gamma$ and thus pulls a \texttt{let}-binding out of the function part of an application.

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

- Finally the “garbage collection” rule \texttt{let-GC} drops a \texttt{let}-binding whose defined variable no longer appears in the term. \texttt{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 $\lambda$-calculus. So $\lambda_{\texttt{let}}$ 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 $\lambda_{\texttt{let}}$ is an extension of the call-by-value $\lambda$-calculus. A $\beta V$-reduction

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

can be expressed by the following sequence of $\lambda_{\texttt{let}}$-reductions $\Gamma$ where there is one \texttt{let-V} step for each occurrence of $x$ in $M$.

$$(\lambda x. M) \ V \quad \texttt{let } x = V \text{ in } M \quad \texttt{let } x = V \text{ in } [V/x] M \quad [V/x] M$$

\textbf{Example 3.1.} $(\lambda x. x) (\lambda y. y)$.

$$(\lambda x. x) (\lambda y. y) \quad \texttt{let } x = \lambda y. y \quad \texttt{let } x = \lambda y. y \quad \texttt{let } x = \lambda y. y \quad [\lambda y. y]$$

\texttt{in } x \quad \texttt{in } (\lambda y. x) \quad \texttt{in } \texttt{let } y = x \quad \texttt{in } y \quad \texttt{in } \texttt{let } y = x \quad \texttt{in } \texttt{let } y = x \quad \texttt{in } \texttt{let } y = x \quad \texttt{in } \lambda y. y \quad \texttt{in } \lambda y. y$$
Graphically we have the following sequence where we mark the node currently considered the root of the graph with a 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 $\lambda$-calculus.

Theorem 4.1. $\lambda_{let}$ is confluent:

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].

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 $\lambda_{let}$ programs. Following the spirit of [Abr90] we define an observation to be a reduction
Additional Syntactic Domains

<table>
<thead>
<tr>
<th>Domain</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Values</td>
<td>$V_s ::= \lambda x.M$</td>
</tr>
<tr>
<td>Answers</td>
<td>$A, A' ::= V_s \mid \text{let } x = M \text{ in } A$</td>
</tr>
<tr>
<td>Evaluation</td>
<td>$E, E' ::= \text{[] \mid E \mid M \mid \text{let } x = M \text{ in } E \mid \text{let } x = E \text{ in } E'[x]$</td>
</tr>
<tr>
<td>Contexts</td>
<td>$E, E' ::= \text{[]} \mid M \mid \text{let } x = M \text{ in } E \mid \text{let } x = E \text{ in } E'[x]$</td>
</tr>
</tbody>
</table>

### Standard Reduction Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Redex</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(let,-I)</td>
<td>$(\lambda x.M) N$</td>
<td>$\Leftrightarrow \text{let } x = N \text{ in } M$</td>
</tr>
<tr>
<td>(let,-V)</td>
<td>$\text{let } x = V_s \text{ in } E[x]$</td>
<td>$\Leftrightarrow \text{let } x = V_s \text{ in } E[V_s]$</td>
</tr>
<tr>
<td>(let,-C)</td>
<td>$(\text{let } x = L \text{ in } A) N$</td>
<td>$\Leftrightarrow \text{let } x = L \text{ in } A N$</td>
</tr>
<tr>
<td>(let,-A)</td>
<td>$\text{let } y = (\text{let } x = L \text{ in } A) \text{ in } E[y]$</td>
<td>$\Leftrightarrow \text{let } x = L \text{ in } \text{let } y = A \text{ in } E[y]$</td>
</tr>
</tbody>
</table>

Figure 3: Standard call-by-need reduction.

sequence that ends in a function term. In $\lambda_{\text{let}} \Gamma$ 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 $\Gamma$

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

has both let's in evaluation position $\Gamma$ 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,-GCI since garbage-collection is not needed to reduce a term to an answer.

**Definition.** Let $\Leftrightarrow$ be the smallest relation that contains $\text{let}_s \{-\text{IVCA}\}$ and that is closed under the implication $M \Leftrightarrow N \Rightarrow E[M] \Leftrightarrow E[N]$.

**Definition.** We write $M \downarrow$ iff there is an answer $A$ such that $M \rightarrow A$. Likewise $\Gamma$ we write $M \downarrow_s$, iff there is an answer $A$ such that $M \Leftrightarrow A$.

**Theorem 4.2.** $\Leftrightarrow$ is a standard reduction relation for $\lambda_{\text{let}}$. For all $M \in \Lambda \Gamma$

$$M \downarrow \Leftrightarrow M \downarrow_s$$
Proof Sketch: The proof relies on two subsidiary results: that all answers are in $\equiv$-normal form and that standard reduction keeps head and internal redexes separate [Bar81Lem- ma 11.4.3]. The result then follows by reasoning as in Barendregt’s standardization proof for the call-by-name calculus. □

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 $\lambda_{let}$ and $\lambda$. The conversion theories $=_{let}$ 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 $\lambda$ and $\lambda_{let}$ coincide on their common term language $\Lambda$.

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

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

**Definition.** $\lambda \vdash M \downarrow$ iff there is an abstraction $\lambda x. N$ such that $\lambda \vdash M \rightarrow \lambda x. N$.

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 $\lambda^* \Gamma$ written $\lambda^* \vdash M \equiv N \Gamma$ iff for all $\lambda^*$-contexts $C$ such that $C[M]$ and $C[N]$ are closed

$$\lambda^* \vdash C[M] \downarrow \iff \lambda^* \vdash C[N] \downarrow.$$ 

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

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

- $x^* = x$
- $(\lambda x. M)^* = \lambda x. M^*$
- $(M \ N)^* = M^* \ N^*$
- $(let \ x = M \ in \ N)^* = [M^*/x]N^*$

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

The following proposition is a direct consequence of the finite developments theorem in $\lambda$-calculus (with $let$ terms as labeled $\beta$-redexes).
Proposition 5.1. Every term in $\lambda_{\text{let}}$ has a let-expansion.

Lemma 5.2. For all terms $M \Gamma N$

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

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

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

$$(C[M])^* \equiv C^*[\sigma M^*, \ldots, \sigma M^n]$$

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

Lemma 5.4. For all terms $M \Gamma N$ if $M \rightarrow N$ 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 $M, N \in \lambda_{\text{let}}$ the following diagram commutes.

\[ \begin{array}{ccc} M & \xrightarrow{\text{let}} & N \\ \downarrow & & \downarrow \\ M^* & \xrightarrow{\beta} & N^* \end{array} \]

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

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

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

$$\begin{array}{c} M \xRightarrow{\text{let}} N \\ C[M] \xRightarrow{\beta} C[N] \end{array}$$

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

$$(C[M])^* \equiv C^*[\sigma M^*, \ldots, \sigma M^n]$$

$$(C[N])^* \equiv C^*[\sigma N^*, \ldots, \sigma N^n].$$

By the induction hypothesis $M^* \xRightarrow{\beta} N^*$. Therefore $((C[M])^* \equiv C^*[\sigma M^*, \ldots, \sigma M^n] \xRightarrow{\beta} C^*[\sigma N^*, \ldots, \sigma N^n] \equiv (C[N])^*$. □
For the proof of the next lemmas we need some more notation and definitions. In the following we will write a term \( \text{let } x_1 = M_1 \text{ in } \ldots \text{ let } x_n = M_n \text{ in } M \) alternatively as \( \text{let } \Phi \text{ in } M \Gamma \) where the heap \( \Phi \) is the sequence of bindings \( (x_1 \mapsto M_1, \ldots, x_n \mapsto M_n)^2 \). In general, heaps are finite sequences of bindings \( (x_i \mapsto M_i) \) such that \( x_i = x_j \Rightarrow i = j \) and \( x_i \in \text{fv}(M_j) \Rightarrow i < j \). Let \( (\Phi_1, \Phi_2) \) denote the concatenation of the heaps \( \Phi_1 \) and \( \Phi_2 \) where is assumed that the bound variables of \( \Phi_1 \) and \( \Phi_2 \) form disjoint sets. Furthermore, \( \Gamma \) let the defining occurrence of a variable \( x \) in a heap \( \Phi \) be

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

Finally, we extend \( \text{let} \)-expansion to a mapping from heaps to substitutions \( \Gamma \) by defining

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

**Lemma 5.6.** For all terms \( M \in \Lambda \text{ let } \Gamma N \in \Lambda \) there exist terms \( M' \in \Lambda \text{ let } \Gamma N' \in \Lambda \) such that the following diagram commutes.

\[
\begin{array}{c}
M \xrightarrow{\text{let}} M' \\
\downarrow^* \quad \downarrow^* \\
M^* \xrightarrow{\beta} N \xrightarrow{\beta} N'
\end{array}
\]

**Proof:** W.l.o.g. assume that the term \( M \) in this diagram is in \( \text{let-V-C-A-GC} \) normal form — if it is not we 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 \( M^* \xleftrightarrow{\beta} N \). There are two cases.

**Case 1** The reduction is a top-level application of the \( \beta \) rule. In this case \( \Gamma \)

\[
M^* \equiv (\lambda x. Q_1)Q_2 \xleftrightarrow{\beta} [Q_1/x]Q_2 \equiv N
\]

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

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

\[
M \equiv \text{let } \Phi_1, y \mapsto (\lambda x. P_1)P_2, \Phi_2 \text{ in } z \\
\equiv \text{let } \Phi_1, y \mapsto \text{let } x = P_2 \text{ in } P_1, \Phi_2 \text{ in } z \\
\equiv \text{let } \Phi_1, x \mapsto P_2, y \mapsto P_1, \Phi_2 \text{ in } z \quad \text{def} \quad M'
\]

\(^2\text{Note the relationship to the natural semantics in Section 7.}\)
Furthermore, \( (M')^* \equiv \Phi^*_1([P_1^* / x]([P_2^* / y](\Phi^*_2 z))) \)
\[ \equiv \Phi^*_1([P_2^* / x]([P_1^* / y]y)) \quad \text{since def}(z, \Phi_2) = y \]
\[ \equiv \Phi^*_1([P_2^* / x]P_1^*) \quad \text{since fvt}([P_2^* / x]P_1^*) \cap (\text{dom}(\Phi_2) \cup \{x, y\}) = \emptyset \]
\[ \equiv [Q_2 / x]Q_1 \quad \text{by Lemma 5.2} . \]

**Case 1.2** \( M_0 \equiv (\lambda x. P_1) P_2 \Gamma \) for some terms \( P_1 \Gamma P_2 \) 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 \( M_0 \equiv \lambda y. PT \Gamma \) for some variable \( y \Gamma \) term \( P \). Then by definition of \((^*)\Gamma M^* \equiv \lambda y. \Phi^* P^* \Gamma \) a contradiction. Or assume that \( M_0 \equiv P_1 P_2 P_3 \) for some terms \( P_1 \Gamma P_2 \Gamma P_3 \). In this case \( M^* \equiv (\Phi^* P_1^*) (\Phi^* P_2^*) (\Phi^* P_3^*) \) a contradiction. Finally assume that \( M_0 \equiv y PT \Gamma \) for some variable \( y \Gamma \) term \( P \). We then distinguish according to \text{def}(y, \Phi). If \text{def}(y, \Phi) \) is a variable we get a contradiction. If \text{def}(y, \Phi) \) is a \( \lambda \)-abstraction we get a contradiction since then \( M \) is not in \text{let-V-C-A-GC} normal form. Finally if \( y \) is an application \( P_1 P_2 \) 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 \( M^* \not\Rightarrow N \) is top-level.

**Case 2** The redex in the reduction \( M^* \not\Rightarrow N \) is a proper subterm of \( M^* \). In this case the last step in the proof of this reduction is an application of the rule

\[
\frac{Q_1 \not\Rightarrow Q'_1}{Q_1 Q_2 \not\Rightarrow Q'_1 Q_2}
\]

where \( M^* \equiv Q_1 Q_2 \). Let again \( M \equiv \text{let} \Phi \) in \( M_0 \) where \( M_0 \) is not a \text{let-binding}. Then one of the following cases applies.

**Case 1.1** \( M_0 \equiv z \Gamma \) for some variable \( z \) and \text{def}(z, \Phi) = P_1 P_2 \Gamma \) for some terms \( P_1 \Gamma P_2 \) with \( \Phi^* P_i^* \equiv Q_i^* \) (\( i = 1, 2 \)). Then there exist heaps \( \Phi_1 \Gamma \Phi_2 \) and a variable \( y \) such that \( \Phi = (\Phi_1, y \mapsto P_1 P_2, \Phi_2) \) and \text{def}(z, \Phi_2) = y. Since \text{fvt}(P_1 P_2) \cap (\text{dom}(\Phi_2) \cup \{y\}) = \emptyset \) we have also that \( \Phi_1^* P_i^* \equiv Q_i^* \) (\( i = 1, 2 \)). By the induction hypothesis let \( \Phi_1 \) in \( P_1 \not\Rightarrow P_1' \Gamma \) for some term \( P_1' \) such that \( Q_1' \not\Rightarrow (P_1')^* \). A straightforward induction on the notion of reduction in \text{let} shows that \( P_1 \) is of the form \text{let} \( \Phi'_1 \) in \( P_1'' \) where for all terms \( P \Gamma \text{let} \Phi_1 \) in \( P \not\Rightarrow \text{let} \Phi'_1 \) in \( P \). It follows that

\[
M \equiv \text{let} \Phi_1, y \mapsto P_1 P_2, \Phi_2 \in z \not\Rightarrow \text{let} \Phi'_1, y \mapsto P_1'' P_2, \Phi_2 \in z \not\Rightarrow M' .
\]
Furthermore
\[ (M')^* \equiv (\Phi'_1)^*([((P''_1 P'_2)^* y])(\Phi''_2 z)) \]
\[ \equiv (\Phi'_1)^*([((P''_1 P'_2)^* y)y]) \quad \text{since } \text{def}(z, \Phi_2) = y \]
\[ \equiv ((\Phi'_1)^*(P''_1)^*)((\Phi'_1)^* P''_2) \]
\[ \equiv Q'_1((\Phi'_1)^* P''_2) \]
\[ \text{def } N' . \]

Since \((\Phi'_1)^* P''_2 \equiv (\text{let } \Phi'_1 \text{ in } P''_2)^* \) and \( \text{let } \Phi_1 \text{ in } P''_2 \rightarrow \text{let } \Phi'_1 \text{ in } P''_2 \Gamma \), it follows with Lemma 5.5 that \( Q_2 \equiv (\text{let } \Phi_1 \text{ in } P''_2)^* \overset{\leftrightarrow}{=} ((\Phi'_1)^* P''_2) . \) In summary \( Q'_1 Q_2 \overset{\leftrightarrow}{=} N' . \)

**Case 2.2** \( M_0 \equiv P_1 P_2 \Gamma \) for some terms \( P_1 \Gamma P_2 \) 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 \( \Gamma \) we can show that no other subcases apply. \( \square \)

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 \( M, N \in \Lambda \Gamma \) such that \( \lambda \vdash M \leftrightarrow N \) and \( \lambda \vdash M \leftrightarrow^n A \Gamma \) for some answer \( A \Gamma n \geq 0 \). Then \( N \leftrightarrow^n A' \) 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 \( M \in \Lambda|_{\text{let}} \Gamma \)
\[ \lambda|_{\text{let}} \vdash M \Downarrow \iff \lambda \vdash M^* \Downarrow . \]

**Proof:** “\( \Rightarrow \)”: An easy induction on the length of reduction from \( M \) to an answer \( \Gamma \) using Lemma 5.5 at each step.

“\( \Leftarrow \)”: Assume that \( M^* \Downarrow . \) By the Curry-Feys standardisation theorem [Bar81 \( \Gamma \) 11.4.7] \( \Gamma \) \( M \) 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 \( \Gamma \) i.e. a \( \lambda \)-abstraction \( \Gamma \) say \( \lambda x. Q \). Then the definition of \( \text{let}-\text{expansion} \) implies that one of two cases apply. Either \( M \equiv \text{let } \Phi \text{ in } \lambda x. P \Gamma \) for some heap \( \Phi \Gamma \) term \( P \) such that \( \Phi^* P^* \equiv Q \). Then \( M \) is an answer in \( \lambda|_{\text{let}} . \) Or \( M \equiv \text{let } \Phi \text{ in } z \Gamma \) for some heap \( \Phi \Gamma \) variable \( z \) such that \( \text{def}(z, \Phi) = \lambda x. P \) and \( \Phi^* P^* \equiv Q \). In that case \( M \) reduces to the answer \( \text{let } \Phi \text{ in } \lambda x. P \) by a sequence of \( \text{let-V} \) reductions.

Assume now that \( M^* \Downarrow^\sim n \) for \( n > 0 \). Let \( N \) be the standard reduce of \( M^* \Gamma \) i.e. \( M^* \leftrightarrow N \) and \( N \Downarrow^\sim n-1 . \) By Lemma 5.6 there is a term \( M' \in \Lambda|_{\text{let}} \) such that \( \lambda|_{\text{let}} \vdash M \rightarrow M' \) and \( N \leftrightarrow (M')^* . \) By Proposition 5.7 \( (M')^* \Downarrow^\sim n-1 \Gamma \) therefore \( \Gamma \) by the induction hypothesis \( M' \Downarrow . \) With \( \lambda|_{\text{let}} \vdash M \rightarrow M' \) this implies that \( M \Downarrow . \) \( \square \)
Proposition 5.8 implies that $\lambda_{let}$ is a conservative observational extension of $\lambda$:

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

$$\lambda \models M \cong N \iff \lambda_{let} \models M \cong N.$$  

**Proof:** "$\Rightarrow$": Assume $\lambda \models M \cong N$ and let $C$ be a $\lambda_{let}$-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

$$\lambda_{let} \downarrow C[M] \downarrow \quad \iff \quad \lambda_{let} \downarrow C#[M] \downarrow \quad \text{since } \lambda_{let} \downarrow C#[M] = C[M]$$
$$\iff \lambda \downarrow C#[M] \downarrow \quad \text{by Proposition 5.8 since } (C#[M])^* = C#[M]$$
$$\iff \lambda \downarrow C#[N] \downarrow \quad \text{since } \lambda \models M \cong N$$
$$\iff \lambda_{let} \downarrow C[N] \downarrow \quad \text{by the reverse argument}$$

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

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

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

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

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

6 The Let-Less Call-By-Need Calculus

In call-by-name $\lambda$-calculus $\Gamma$ let $x = M$ in $N$ is syntactic sugar for $(\lambda x. M)N$; 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 $\lambda_\ell$ (where the $\ell$ stands for "lazy"). Its notions of general and standard reduction are shown in Figure 4.

While $\lambda_\ell$ is perhaps somewhat less intuitive than $\lambda_{let}$, 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 $\lambda_{let}$ and $\lambda_\ell$ are closely related. More precisely, the following theorem states that reduction in $\lambda_{let}$ can be simulated in $\lambda_\ell \Gamma$ and that the converse is also true, provided we identify terms that are equal up to let-I introduction.

**Proposition 6.1.** For all $M \in \Lambda_\ell \Gamma M' \in \Lambda_{let} \Gamma$
Proposition 6.1 can be used to derive the essential syntactic and properties of $\lambda_{\ell}$ from those of $\lambda_{\text{let}}$:

**Theorem 6.2.** $\lambda_{\ell}$ is Church Rosser.

**Theorem 6.3.** $\cong$ is a standard reduction relation for $\lambda_{\ell}$. For all $M \in \Lambda_{\ell}$

$$M \Downarrow_{\ell} \cong M \Downarrow_{s}.$$  

$\lambda_{\ell}$ has close relations to both the call-by-value calculus $\lambda_{V}$ and the call-by-name calculus $\lambda$. Its notion of equality $\equiv_{\lambda_{\ell}}$ — i.e., the least equivalence relation generated by the reduction relation — fits between those of the other two calculi making $\lambda_{\ell}$ an extension of $\lambda_{V}$ and $\lambda$ an extension of $\lambda_{\ell}$.
Theorem 6.4.

\[ \Rightarrow_{\lambda V} \subseteq \Rightarrow_{\lambda} \subseteq \Rightarrow_{\lambda \ell} \cdot \]

**Proof:** (1) \( \beta V \) can be expressed by a sequence of \( \lambda_{\ell} \) reductions as was shown at the end of Section 3. Therefore \( \Gamma \Rightarrow_{\lambda V} \subset \Rightarrow_{\lambda} \). (2) Each \( \lambda_{\ell} \) reduction rule is an equality in \( \lambda \). For instance in the case of \( \ell-V \) one has:

\[ (\lambda x.C[x]) V \Rightarrow_{\beta} [V/x](C[x]) \equiv [V/x](C[V]) \Rightarrow_{\beta} (\lambda x.C[V]) V \]

The other rules have equally simple translations. In summary \( \Gamma \Rightarrow_{\lambda_{\ell}} \subset \Rightarrow_{\lambda} \). \( \square \)

Each of the inclusions of Theorem 6.4 is proper; e.g., \( \Gamma \)

\[ (\lambda x.x) ((\lambda y.y) \Omega) = (\lambda y.((\lambda x.x) y) \Omega \]

is an instance of rule \( \ell-\Delta \Gamma \) but it is not an equality in the call-by-value calculus (\( \Omega \) stands for a non-terminating computation). On the other hand \( \Gamma \) the following instance of \( \beta \) is not an equality in \( \lambda_{\ell} \):

\[ (\lambda x.x) \Omega = \Omega \cdot \]

However \( \Gamma \) one can show by a simple application of Theorem 5.9 together with Proposition 6.1 that the observational equivalence theories of \( \lambda_{\ell} \) and \( \lambda \) are identical (and are incompatible with the observational equivalence theory of \( \lambda V \)).

**Theorem 6.5.** For all terms \( M, N \in \Lambda \Gamma \)

\[ \lambda \models M \equiv N \iff \lambda_{\ell} \models M \equiv N. \]

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

### 7 Natural semantics

This section presents an operational semantics for call-by-need in the natural semantics style of Plotkin and Kahn\( \Gamma \) 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 \( \Gamma \)

\[ x_1 \mapsto M_1, \ldots, x_n \mapsto M_n. \]

The order of the sequence of bindings is significant: all free variables of a term must be bound to the left of it. Furthermore \( \Gamma \) all variables bound by the heap must be distinct.
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 \( 1 \leq i \leq n \Gamma \) and all the \( x_i \) are distinct. Let \( \Phi, \Psi, \Upsilon \) range over heaps. If \( \Phi \) is the heap \( x_1 \mapsto M_1, \ldots, x_n \mapsto M_n \Gamma \) define \( \text{vars}(\Phi) = \{x_1, \ldots, x_n\} \). A configuration pairs a heap with a term \( \Gamma \) where the free variables of the term are bound by the heap. Thus \( \langle \Phi \rangle M \) is well-formed if \( \Phi \) 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 \( \Gamma \) the heap is left unchanged and the abstraction is returned.

- **Applications are straightforward.** Evaluate the function to yield a lambda abstraction \( \Gamma \) extend the heap so that the the bound variable of the abstraction is bound to the argument \( \Gamma \) then evaluate the body of the abstraction. In this rule \( x' \) is a new name not appearing in \( \Psi \) 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 \( \Gamma \) evaluate the term \( \Gamma \) 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 \( \Phi \) is required to evaluate \( M \). Evaluation yields a new heap \( \Psi \) and value \( V \). The new heap \( \Psi \) will differ from the old heap \( \Phi \) in two ways: binding may be updated (by \( \text{Var} \)) and bindings may be added (by \( \text{App} \)). The free variables of \( V \) are bound by \( \Psi \Gamma \) so to ensure the heap stays well-formed the final heap has the form \( \Psi, x \mapsto V, \Upsilon \).

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

As one would expect evaluation uses only well-formed configurations \( \Gamma \) and evaluation only extends the heap.
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 \( \text{vars}(\Phi) \subseteq \text{vars}(\Psi) \).

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 \( \Phi \) is the heap \( x_1 \mapsto M_1, \ldots, x_n \mapsto M_n \), write \( \text{let } \Phi \text{ in } N \) for the term

\[
\text{let } x_1 = M_1 \text{ in } \cdots \text{ let } x_n = M_n \text{ in } N.
\]

Every answer \( A \) can be written \( \text{let } \Psi \text{ in } V \) for some heap \( \Psi \) 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 \( \lambda \Downarrow \text{let } \Phi \text{ in } M \leftrightarrow \text{let } \Psi \text{ in } V \).

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 \( \lambda_{\text{let}} \) 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 \( \text{inl} (1+1) \) is not a legal value since copying it would also copy the unevaluated term \( (1+1) \). Instead one writes \( \text{let} x = 1+1 \text{ in } \text{inl} x. \)

There are two new reduction rules. Rule \( \delta-V \) is the usual rewrite rule for primitive operator application. It is defined in terms of a partial function — also called \( \delta \) — 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 \( M \Gamma \delta(p,C[\lambda x.M]) = D[\lambda x.M] \) or \( \delta(p,C[\lambda x.M]) \) is undefined. Note that rule \( \delta-V \) makes all primitive operators unary and strict. Operators of more than one argument can still be simulated by currying. Rule \( \delta-A \) allows \( \text{let } \Leftrightarrow \text{bindings of operator arguments to be pulled out of the application.} \)

**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 \( \lambda_{\Gamma} \) indirection shortening is a sort of non-standard \( \ell-V \) redex: namely the case where the argument is a single free variable.

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

Let \( \lambda^S_{\ell} \) be the theory obtained by extending \( \lambda_{\ell} \) with \( \ell-S \).

The extension does not alter the resulting theory:

**Theorem 8.1.**

\[
\lambda^S_{\ell} \models M \triangleq N \Leftrightarrow \lambda_{\ell} \models M \triangleq N.
\]

**Proof:** Trivially by Theorem 5.9 since \( \ell-S \) contractions are just ordinary \( \beta \) contractions.

\( \square \)

Likewise a rule for indirection shortening will not alter the theory \( \lambda_{\text{let}} \).
Although Launchbury does not identify an indirection shortening rule such as an extension for his system would be relatively simple:\n\[ \frac{[y/x] \Gamma : \epsilon \Downarrow \Delta : z}{(\Gamma, x \mapsto y) : \epsilon \Downarrow \Delta : z, \ x \notin \text{fv}(\Gamma)}. \]

\( \ell \)-GC and \( \ell \)-S allow a nice two-layered generalization of the standard ordering. At one layer is \( \ell \_\text{-VICIA} \) by itself deterministic and which will therefore always reach answers when they exist. At a second layer \( \ell \_\text{-GC} \) reduction \( \Gamma \) while not deterministic is strongly normalizing \( \Gamma \) and the two layers together form a confluent calculus which \( \Gamma \) like \( \ell \_\text{-VICIA} \) 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 \( \lambda \). 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)
\[ \text{let } x \equiv (1+1) : x \text{ in } x. \]

It seems possible to extend our calculus with a recursive \textbf{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

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

\(^3\) with the notable exception of [Lau93].


A Proofs

A.1 Church-Rosser

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

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

where $i$ is a positive integer.

Let $\iota$ be a map from the individual variables of a term $T \in \Lambda^*_\text{let}$ to positive integers; then $(T, \iota)$ denotes the $\Lambda^*_\text{let}$ term obtained by weighting $T$’s variables according to $\iota$.

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

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

Proof: Trivial; each term’s weights define the appropriate function. □

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

$$\|x^i\| = i + 1$$
$$\|e_1 e_2\| = \|e_1\| \cdot \|e_2\|$$
$$\|\lambda x.e\| = 2 + \|e\|$$
$$\|\text{let } x = e_1 \text{ in } e_2\| = \|e_1\| \cdot \|e_2\| + 1 .$$
Lemma A.4. Let \( T \in \Lambda_{\text{let}}^* \). Then \( \|T^*\| > 1 \).

Proof: Trivial. \( \square \)

Definition A.5. A term \( T^* \in \Lambda_{\text{let}}^* \) is said to have a decreasing weighting (or for a term \((T, i)\), that \( i \) is a decreasing weighting) if for every subterm \( \text{let } x = M^* \) 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 occurrences in \( T \) from two to right to left, except that within an expression \( \text{let } x = M \) 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]. \( \square \)

Lemma A.8. Let \( T^* \in \Lambda_{\text{let}}^* \) and

\[ T^* \xrightarrow{\text{let-\{I,C,A,GC\}}} T'' . \]

Then \( \|T''\| < \|T^*\| . \)

Proof: Trivial, by analysis of the before-and-after term structures. \( \square \)

Lemma A.9. Let \( T^* \equiv (T, i) \in \Lambda_{\text{let}}^* \) a decreasing weighting for \( T \), and

\[ T^* \equiv (T, i) \xrightarrow{\text{let-\{V,C,A,GC\}}} T'' . \]

Then \( \|T''\| < \|T^*\| . \)

Proof: Again, by analysis of the terms. \( \square \)

Corollary A.10. Let \( T^* \equiv (T, i) \in \Lambda_{\text{let}}^* \) a decreasing weighting for \( T \), and

\[ T^* \equiv (T, i) \xrightarrow{\text{let-\{V,C,A,GC\}}} T'' . \]

Then \( \|T''\| < \|T^*\| . \)

Proof: Follows from Lemma A.8 and Lemma A.9. \( \square \)

Lemma A.11. Let \( T \in \Lambda_{\text{let}}^* \) a decreasing weighting for \( T \), and

\[ \xrightarrow{\text{let-\{V,C,A,GC\}}} (T', i') . \]

Then \( i' \) is a decreasing weighting for \( T' \).

Proof: As in [Bar81, Lemma 11.2.18(ii)]. \( \square \)
Lemma A.12.

\[
\begin{array}{c}
M \\
\text{let-I} \\
M' \\
\text{let-I} \\
\exists N \\
\end{array}
\]

\[
\begin{array}{c}
M' \\
\text{let-I} \\
M'' \\
\text{let-I} \\
\exists N \\
\end{array}
\]

Proof: By a trivial structural induction. □

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

\[
\begin{array}{c}
M \\
\text{let-I} \\
M' \\
\text{let-I} \\
\exists N \\
\end{array}
\]

\[
\begin{array}{c}
M'' \\
\text{let-I} \\
\exists N \\
\end{array}
\]

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. □


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]. □


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

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

\[
\begin{array}{c}
M \\
\text{let} \\
M' \\
\text{let} \\
\exists N \\
\end{array}
\]

\[
\begin{array}{c}
M'' \\
\text{let} \\
\exists N \\
\end{array}
\]

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 $M$, $x \in \text{fv}(M)$, 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:

\[ E_0[x] \equiv C[x] \quad C[V] \equiv E[x] \]

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

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 contracted from those subterms.

Lemma A.21. Let $A \in A$. Then $A \not\rightarrow_\beta$.

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: $A \equiv \text{let } x = M \text{ in } A'$. Follows from the inductive hypothesis and Lemma A.20.a.

Lemma A.22. Let $M \xrightarrow{\lambda_{\text{let}\setminus\lambda_{\text{let}}}} A$. Then $M \in A$.

Proof: By structural induction over $A$.

Case 1: $A \equiv \lambda x. N$. Clearly $A$ is not itself the contraction of any $\text{let-}\{\text{I,V,C,A}\}$-redex, so two cases are possible:

Case 1.A: $M \xrightarrow{\lambda_{\text{let}\setminus\lambda_{\text{let}}}} A$. Then $M \equiv \text{let } y = M' \text{ in } A \in A$.

Case 1.B: $M \equiv \lambda x. M', M' \xrightarrow{\lambda_{\text{let}\setminus\lambda_{\text{let}}}} N$. Clearly $M \in A$.

Case 2: $A \equiv \text{let } x = N \text{ in } A'$. By analysis of the position of the redex within $M$:

Case 2.A: $\langle M, A \rangle \in \lambda_{\text{let}} \setminus \lambda_{\text{let}}$. $M$ cannot be a $\text{let}$-redex, as all top-level $\text{let}$-redexes are standard redexes. We also cannot have $M$ a $\text{let}$-$\text{C}$-redex, which always produces an application, and hence never an answer.

Case 2.A.1: $\langle M, A \rangle \in \text{let-V} \setminus \text{let-V}$.

Case 2.A.2: $\langle M, A \rangle \in \text{let-A} \setminus \text{let-A}$.

Case 2.A.3: $\langle M, A \rangle \in \text{let-GC} \setminus \text{let-GC}$.

Case 2.B: $M \equiv \text{let } x = M' \text{ in } M''$, and one of the following:

Case 2.B.1: $M' \xrightarrow{\lambda_{\text{let}\setminus\lambda_{\text{let}}}} N$, $M'' \equiv A$, and so $M \in A$.

Case 2.B.2: $M'' \xrightarrow{\lambda_{\text{let}\setminus\lambda_{\text{let}}}} A$. So by the induction hypothesis $M'' \in A$, and then $M'' \in A$.

\[ \square \]
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 \( M \xrightarrow{\lambda_{\text{let}} \setminus \lambda_{\text{let}} - \text{let,} \text{-} \text{I}} N \xrightarrow{\text{let,} \text{-} \text{I}} N' \). Then there exists a term \( M' \) such that \( M' \xrightarrow{\lambda_{\text{let}} \setminus \lambda_{\text{let}} - \text{let,} \text{-} \text{I}} N' \) and exactly one of the following is true:

1. \( M \xrightarrow{\text{let,} \text{-} \text{I}} M' \).

2. There exists another term \( M'' \) such that \( M \xrightarrow{\text{let,} \text{-} \text{C}} M'' \xrightarrow{\text{let,} \text{-} \text{I}} M' \).

**Proof:** By several case analyses:

Case 1: \( \langle M, N \rangle \in \lambda_{\text{let}} \setminus \lambda_{\text{let}} , \text{-} \text{I} \). We cannot have \( \langle M, N \rangle \in \text{let,} \text{-} \text{I} \setminus \lambda_{\text{let}} , \text{-} \text{I} \), since all top-level let-I-redexes are standard.

**Case 1.A:** \( \langle M, N \rangle \in \text{let,} \text{-} \text{I} \setminus \lambda_{\text{let}} , \text{-} \text{I} \). So let

\[
M \equiv \text{let } x = V \text{ in } C[x], \\
N \equiv \text{let } x = V \text{ in } C[V] \equiv \text{let } x = V \text{ in } E[(\lambda y . L_1) \; L_2], \\
N' \equiv \text{let } x = V \text{ in } E[\text{let } y = L_2 \text{ in } L_1],
\]

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

\[
M \equiv \text{let } x = V \text{ in } C[x] \xrightarrow{\text{let,} \text{-} \text{I}} N \equiv \text{let } x = V \text{ in } C[V] \equiv \text{let } x = V \text{ in } (\lambda y . L_1) \; L_2 \xrightarrow{\text{let,} \text{-} \text{I}} N' \equiv \text{let } x = V \text{ in } \text{let } y = L_2 \text{ in } L_1
\]

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

\[
M \equiv \text{let } x = V \text{ in } (\lambda y . C_1[x]) \; L_2 \xrightarrow{\text{let,} \text{-} \text{I}} M' \equiv \text{let } x = V \text{ in } \text{let } y = L_2 \text{ in } C_1[x] \xrightarrow{\text{let,} \text{-} \text{I}} \text{let } x = V \text{ in } \text{let } y = L_2 \text{ in } C_1[V] \equiv N'
\]

And similarly for \( V \) a subterm of \( L_1 \).

Case 1.A.2: \( E \equiv E_1 \; N_3 \). So we have that \( C \) must also be an application; if we have \( C \equiv E_1[(\lambda x . L_1) \; L_2] \; C_3 \) then the result is trivial. Otherwise, for \( C \equiv C_1 \; N_3 \) 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: \( E \equiv \text{let } z = N_1 \text{ in } E_1 \), and

Case 1.A.4: \( E \equiv \text{let } z = E_2 \text{ in } E_1[z] \). Similarly.

**Case 1.B:** \( \langle M, N \rangle \in \text{let,} \text{-} \text{A} \setminus \lambda_{\text{let}}, \text{-} \text{A} \). So let

\[
M \equiv \text{let } x = (\text{let } y = M_0 \text{ in } M_1) \text{ in } M_2, \\
N \equiv \text{let } y = M_0 \text{ in } \text{let } x = M_1 \text{ in } M_2,
\]

25
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

\[
N' \equiv \text{let } y = M'_0 \text{ in let } x = M_1 \text{ in } M_2
\]

(\( \text{let } y = M_0 \text{ in let } x = M'_1 \text{ in } M_2 \) and \( \text{let } y = M_0 \text{ in let } x = M_1 \text{ in } M'_2 \)), and so

\[
M' \equiv \text{let } x = (\text{let } y = M'_0 \text{ in } M_1) \text{ in } M_2
\]

(\( \text{let } y = M_0 \text{ in let } x = M'_1 \text{ in } M_2 \) and \( \text{let } x = (\text{let } y = M_0 \text{ in } M_1) \text{ in } M'_2 \)).

**Case 1.C:** \( \langle M, N \rangle \in \text{let-C} \setminus \text{let-N-C} \). As in case 1.B.

**Case 1.D:** \( \langle M, N \rangle \in \text{let-GC} \). So \( M \equiv \text{let } x = M_0 \text{ in } N, x \not\in \text{fv}(N) \), and \( M' \equiv \text{let } x = M_0 \text{ in } N' \).

**Case 2:** \( \langle M, N \rangle \not\in \text{let-I} \).

**Case 2.A:** \( \langle N, N' \rangle \in \text{let-I} \). So we have \( N \equiv (\lambda x. N_1) N_2 \), and we distinguish two subcases:

**Case 2.A.1:** \( M \equiv (\text{let } y = N_0 \text{ in } \lambda x. N_1) N_2 \), \langle \text{let } y = N_0 \in \lambda x. N_1, \lambda x. N_1 \rangle \in \text{let-GC} \). \( \lambda x. N_1 \in \mathcal{A} \), so we have

\[
(\text{let } y = N_0 \text{ in } \lambda x. N_1) N_2
\]

\[
\frac{\text{let-C\ } \text{let } y = N_0 \in (\lambda x. N_1) N_3}{\text{let-I\ } \text{let } y = N_0 \text{ in let } x = N_2 \text{ in } N_1}
\]

\[
\frac{\text{let-GC\ } \text{let } x = N_2 \text{ in } N_1 \equiv N'}{\text{let-I\ } \text{let } y = N_0 \text{ in let } x = N_2 \text{ in } N_1}
\]

**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 \( M' \equiv \text{let } x = N'_0 \text{ in } N'_1 \) where of the two pairs \( N'_1 \) & \( N_1 \) and \( N'_2 \) & \( N_2 \), in one pair the first reduces to the second, and in the other, the two are identical.

**Case 2.B:** \( \langle N, N' \rangle \not\in \text{let-I} \). 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:** \( N' \equiv N'_0 M_3, M_0 \xrightarrow{\text{let-I\ } \text{let } y N_0 \text{ in let } x = N_2 \text{ in } N_1} N'_1 \), and

**Case 2.B.1.b:** \( N' \equiv M_0 N'_2, M_3 \xrightarrow{\text{let-I\ } \text{let } y N_0 \text{ in let } x = N_2 \text{ in } N_1} N'_3 \). By the induction hypothesis.

**Case 2.B.1.c:** \( N' \equiv M'_0 M'_2, M_0 \xrightarrow{\text{let-I\ } \text{let } y N_0 \text{ in let } x = N_2 \text{ in } N_1} M'_1, M_3 \xrightarrow{\text{let-I\ } \text{let } y N_0 \text{ in let } x = N_2 \text{ in } N_1} M'_2 \), and

**Case 2.B.1.d:** \( N' \equiv M'_0 M'_2, M_0 \xrightarrow{\text{let-I\ } \text{let } y N_0 \text{ in let } x = N_2 \text{ in } N_1} M'_1, M_3 \xrightarrow{\text{let-I\ } \text{let } y N_0 \text{ in let } x = N_2 \text{ in } N_1} M'_2 \). Trivial.

**Case 2.B.2:** \( M \equiv \text{let } x = M_0 \text{ in } M_3 \). Four similar subcases, each either by the induction hypothesis or trivial subterm analysis.

\( \square \)

**Lemma A.24.**

\[
M \xrightarrow{\lambda x. \setminus, \lambda x. \setminus} N
\]

\[
\cdot \xrightarrow{\text{let-I\ } \text{let } y N_0 \text{ in let } x = N_2 \text{ in } N_1}
\]

\[
\cdot \xrightarrow{\text{let-I\ } \text{let } y N_0 \text{ in let } x = N_2 \text{ in } N_1}
\]

\[
\cdot \xrightarrow{\text{let-I\ } \text{let } y N_0 \text{ in let } x = N_2 \text{ in } N_1}
\]

\[
\cdot \xrightarrow{\text{let-I\ } \text{let } y N_0 \text{ in let } x = N_2 \text{ in } N_1}
\]

**Proof:** By several case analyses:

**Case 1:** \( \langle M, N \rangle \in \lambda x. \setminus, \lambda x. \setminus \). We again cannot have \( \langle M, N \rangle \in \text{let-I\ } \text{let } y N_0 \text{ in let } x = N_2 \text{ in } N_1 \), since all top-level \text{let-I-redexes} are standard.

26
Case 1.A: \( \langle M, N \rangle \in \text{let-} V \backslash \text{let-} V \). So

\[
\begin{align*}
    M & \equiv \text{let } x = V \text{ in } C[x] , \\
    N & \equiv \text{let } x = V \text{ in } C[V] , \\
\end{align*}
\]

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: \( \langle M, N \rangle \in \text{let-} A \backslash \text{let-} A \). So

\[
\begin{align*}
    M & \equiv \text{let } x = (\text{let } y = M_0 \text{ in } M_1) \text{ in } M_2 , \\
    N & \equiv \text{let } y = M_0 \text{ in } \text{let } x = M_1 \text{ in } M_2 , \\
\end{align*}
\]

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

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

\[
M' \equiv \text{let } x = (\text{let } y = M_0 \text{ in } M_1) \text{ in } M_2' .
\]

Case 1.B.2: \( M_2 \equiv E[x], \ M_1 \not\in \mathcal{A} \). 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

\[
M' \equiv \text{let } x = (\text{let } y = M_0 \text{ in } M'_1) \text{ in } M_2 ,
\]

and in the latter case,

\[
M' \equiv \text{let } x = (\text{let } y = M'_0 \text{ in } M_1) \text{ in } M_2 .
\]

Case 1.B.3: \( M_1 \not\in \mathcal{A}, (\overline{A}E) M_2 \equiv E[x] \). As in Case 1.B.1 above.

Case 1.C: \( \langle M, N \rangle \in \text{let-} C \backslash \text{let-} C \). So we have \( M \equiv (\text{let } x = M_0 \text{ in } M_1) \ M_2, \ M_1 \not\in \mathcal{A} \); clearly the standard redex is internal to \( M_1 \), \( M_1 \not\in \text{let-} V \ M'_1 \); and we have

\[
M' \equiv (\text{let } x = M_0 \text{ in } M'_1) \ M_2 .
\]

Case 1.D: \( \langle M, N \rangle \in \text{let-} GC \). So \( M \equiv \text{let } x = M_0 \text{ in } N, \ x \not\in \text{fv}(N) \), and \( M' \equiv \text{let } x = M_0 \text{ in } N' \).

Case 2: \( \langle M, N \rangle \not\in \lambda_{\text{let}} \backslash \lambda_{\text{let}} \).

Case 2.A: \( \langle N, N' \rangle \in \text{let-} - V \). So \( N \equiv \text{let } x = V \text{ in } E[x] \). 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: \( \langle N, N' \rangle \not\in \text{let-} - V \). As in Case 2.B of Lemma A.23.

\[
\square
\]

Lemma A.25.
Proof: By several case analyses:

Case 1: \( \langle M, N \rangle \in \text{let} \setminus \text{let} - \text{let} \). We again cannot have \( \langle M, N \rangle \in \text{let-} \setminus \text{let}, \text{-I} \), since all top-level let-I-redexes are standard.

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

\[
M \equiv \text{let } x = V \text{ in } C[x], \\
N \equiv \text{let } x = V \text{ in } C[V],
\]

where \( C \) is not an evaluation context. As in case 1.A, Lemma A.23.

Case 1.B: \( \langle M, N \rangle \in \text{let-I} \setminus \text{let-A} \). So

\[
M \equiv \text{let } x = (\text{let } y = M_0 \text{ in } M_1) \text{ in } M_2, \\
N \equiv \text{let } y = M_0 \text{ in let } x = M_1 \text{ in } M_2,
\]

where \( M \) is not a standard redex. We again have three cases about the non-standardness of \( M \)'s contraction.

Case 1.B.1: \( M_1 \in A, (\beta E) M_2 \equiv E[x] \). As in Case 1.B.1 of Lemma A.24.

Case 1.B.2: \( M_2 \equiv E_0[x], M_1 \notin A \).

Case 1.B.2.a: \( M_1 \equiv E_1[y] \).

Case 1.B.2.ai: \( M_0 \in V \). So we have

\[
M \equiv \text{let } x = \text{let } y = \lambda z. M_3 \\
\text{in } E_1[y] \\
\text{in } E_0[x], \text{ and}
\]

\[
M \xrightarrow{\text{let-A}} M'' \equiv \text{let } x = \text{let } y = \lambda z. M_3 \\
\text{in } E_1[\lambda z. M_3] \\
\text{in } E_0[x].
\]

If \( E_1[\lambda z. M_3] \) is then an answer, then we have \( M'' \xrightarrow{\text{let-A}} M' \equiv N' \), otherwise \( M'' \equiv M' \).

Case 1.B.2.aii: \( M_0 \in A \setminus V \). So we have

\[
M \equiv \text{let } x = \text{let } y = (\text{let } z = M_0 \text{ in } A_0) \\
\text{in } E_1[y] \\
\text{in } E_0[x], \text{ and}
\]

\[
M \xrightarrow{\text{let-A}} M' \equiv \text{let } x = \text{let } z = M_3 \\
\text{in let } y = A_0 \\
\text{in } E_1[y] \\
\text{in } E_0[x].
\]
and a particularly nasty diagram chase,

\[
\begin{array}{c}
M \equiv \text{let } x = \text{let } y = \text{let } z = M_3 \quad \text{in } A_0 \\
\text{in } E_0[x] \\
\text{let } \lambda_{\text{let-A}} \downarrow \\
M' \equiv \text{let } z = M_3 \quad \text{in } \lambda_{\text{let-A}} \left( N' \equiv \text{let } y = \text{let } z = M_3 \quad \text{in } A_0 \\
\text{in } E_0[x] \right).
\end{array}
\]

Case 1.B.2.a.iii: \( M_0 \not\in A \). Then the standard contraction is clearly internal to \( M_0 \), and \( M' \) is trivially constructed.

Case 1.B.2.b: \((\not\in E_1) M_1 \equiv E_1[y]\). Trivially, the standard redex is a subterm of \( M_0 \), and we have the result by a simple diagram chase.

Case 1.B.3: \( M_1 \not\subseteq A \), \((\not\in E) M_2 \equiv E[x]\). As in Case 1.B.1 of Lemma A.24.

Case 1.C: \( \langle M, N \rangle \in \text{let-C} \setminus \lambda_{\text{let-A}} \). As in Case 1.C of Lemma A.24.

Case 1.D: \( \langle M, N \rangle \in \text{let-GC} \). So \( M \equiv \text{let } x = M_0 \in N \), \( x \not\in \text{fv}(N) \), and \( M' \equiv \text{let } x = M_0 \) in \( N' \).

Case 2: \( \langle M, N \rangle \not\subseteq A \). As in Case 2.B of Lemma A.23.

\( \square \)

**Corollary A.26.**

\[
\begin{array}{c}
M \quad \lambda_{\text{let-A}} \downarrow \\
\vdots \\
\lambda_{\text{let-A}} \{-V,A\} \downarrow \\
M' \quad \lambda_{\text{let-A}} \{-V,A\} \downarrow \\
N \quad N'.
\end{array}
\]

**Proof:** By induction over the length of the reduction sequence from \( M \) to \( N \). \( \square \)

**Lemma A.27.** Let \( M \quad \lambda_{\text{let-A}} \downarrow N \quad \lambda_{\text{let-A}} \downarrow N' \). Then one of the following must be true:

1. There exists some term \( M' \) such that
   \[
   M \quad \lambda_{\text{let-A}} \downarrow M' \quad \lambda_{\text{let-A}} \downarrow N'.
   \]
2. There exists terms $M'$ and $N''$ such that both

$$M \xrightarrow{\text{let}, C} M' \xrightarrow{\lambda_{\text{let}} \setminus \lambda_{\text{let}}'} N''$$

and

$$N' \xrightarrow{\text{let}, C} N''$$.

Proof: By several case analyses:

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

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

$$M \equiv \text{let } x = V \text{ in } C[x]$$,

$$N \equiv \text{let } x = V \text{ in } C[V]$$,

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

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

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

$$N \equiv \text{let } y = M_0 \text{ in } \text{let } x = M_1 \text{ in } M_2$$,

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

$$M \equiv (\text{let } x = M_0 \text{ in } M_1) M_2$$, $M_1 \not\in A$

$$N \equiv \text{let } x = M_0 \text{ in } M_1 M_2$$.

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

Case 1.D: $\langle M, N \rangle \not\in \lambda_{\text{let}} \setminus \lambda_{\text{let}}'$. So $M \equiv \text{let } x = M_0 \text{ in } N$, $x \not\in \text{fv}(N)$, and $M' \equiv \text{let } x = M_0 \text{ in } N'$, $N' \equiv N''$.

Case 2: $\langle M, N \rangle \not\in \lambda_{\text{let}} \setminus \lambda_{\text{let}}'$. Case 2.A: $\langle N, N' \rangle \in \text{let}, C$. So $N \equiv N_1 N_2$, $N_1 \equiv \text{let } x = N_3 \text{ in } A$ and $N' \equiv \text{let } x = N_3 \text{ in } A N_2$.

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

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

$$N_4 N_2 \equiv M \xrightarrow{\lambda_{\text{let}} \setminus \lambda_{\text{let}}'} N \equiv N_1 N_2$$

$$\equiv (\text{let } x = N_3 \text{ in } A) N_2$$

$$\xrightarrow{\text{let}, C} N' \equiv \text{let } x = N_3 \text{ in } A N_2$$.
We have two cases depending on whether $N_4$ is itself the contractum.
Case 2.A.2.a: $\langle N_4, N_1 \rangle \in \lambda_{\text{let}} \setminus \{\lambda_{\text{let}}\}$. By analysis of the specific form of reduction. Note that we cannot have $\langle N_4, N_1 \rangle \in \lambda_{\text{-I}} \setminus \lambda_{\text{-I}}$; such would be a standard redex. We also cannot have $\langle N_4, N_1 \rangle \in \lambda_{\text{-C}} \setminus \lambda_{\text{-C}}$ since $N_1$ would not be an answer, and hence $\langle N, N' \rangle$ not a standard redex.
Case 2.A.2.a.i: $\langle N_4, N_1 \rangle \in \lambda_{-V} \setminus \lambda_{-V}$. So $N_4 \equiv \text{let } x = N_3$ in $A_0$ (we have $A_0 \in \mathcal{A}$ by Lemma A.22) and $A_0 \xrightarrow{\lambda_{-V} \setminus \lambda_{-V}} A$; then:

$$M \equiv N_4 N_2 \xrightarrow{\lambda_{-C}} M' \equiv \text{let } x = N_3 \text{ in } A_0 \quad N' \equiv N' .$$

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

$$A \equiv \text{let } y = M_6 \quad N_4 \equiv \text{let } y = \text{let } x = N_3 \text{ in } M_6 ,$$

and then

$$M \equiv (\text{let } y = \text{let } x = N_3 \text{ in } M_6) \xrightarrow{\lambda_{-A} \setminus \lambda_{-A}} N \equiv (\text{let } x = N_3 \text{ in } M_6) \xrightarrow{\lambda_{-C}} N' \equiv \text{let } x = N_3 \text{ in } \langle \text{let } y = M_6 \text{ in } A_0 \rangle \quad \text{let } y = M_6 \text{ in } \langle \text{let } y = M_6 \text{ in } A_0 \rangle \text{ let } y = M_6 \text{ in } \langle \text{let } y = M_6 \text{ in } A_0 \rangle \text{ let } y = M_6 \text{ in } \langle \text{let } y = M_6 \text{ in } A_0 \rangle .$$

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

$$M \equiv (\text{let } y = M_7 \text{ in } A) \xrightarrow{\lambda_{-G C}} N \equiv (\text{let } x = N_3 \text{ in } A) \xrightarrow{\lambda_{-C}} N' \equiv \text{let } x = N_3 \text{ in } (\text{let } x = N_3 \text{ in } A) \text{ let } x = N_3 \text{ in } (\text{let } x = N_3 \text{ in } A) \text{ let } x = N_3 \text{ in } (\text{let } x = N_3 \text{ in } A) \text{ let } x = N_3 \text{ in } (\text{let } x = N_3 \text{ in } A) .$$
Case 2.A.2.b: \(\langle N_4, N_1 \rangle \not\in \lambda_{\text{let}} \setminus \lambda_{\text{let}}\). So \(N_4 \equiv \text{let } x = N_5 \text{ in } N_6\) where one of the following is true:
Case 2.A.2.b.i: \(N_5 \xrightarrow{\lambda_{\text{let}} \setminus \lambda_{\text{let}}} N_3\) and \(N_6 \equiv A\). Trivially, \(M' \equiv \text{let } x = N_5 \text{ in } (A N_3), N' \equiv N''\).
Case 2.A.2.b.ii: \(N_6 \xrightarrow{\lambda_{\text{let}} \setminus \lambda_{\text{let}}} A\) and \(N_5 \equiv N_5\). Then by Lemma A.22, \(N_6 \in \mathcal{A}\), and we have \(M' \equiv \text{let } x = N_5 \text{ in } (N_6 N_3), N' \equiv N''\).

Case 2.B: \(\langle N, N' \rangle \not\in \lambda_{\text{let}}\). We have \(N' \equiv N''\), with two possible cases for the structure of \(M\) as in Case 2.B of Lemma A.23.

\[\Box\]

Corollary A.28.

\[\begin{array}{c}
M \xrightarrow{\lambda_{\text{let}} \setminus \lambda_{\text{let}}} N \\
\vdots \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2cm} \hspace{2c
Proof: Follows from Corollary A.26, Corollary A.28 and Corollary A.29. We need not worry about a trailing let,-C-sequence since we know \( N' \) to be in \( \lambda_{let,-} \)-normal form. \( \square \)

Lemma A.31. For \( N' \overset{\lambda_{let,-}}{\longrightarrow} \),

\[
\begin{array}{c}
M \overset{\lambda_{let,-}}{\longrightarrow} N \\
\vdots \\
M' \overset{\lambda_{let,-}}{\longrightarrow} N' \overset{\lambda_{let,-}}{\longrightarrow}
\end{array}
\]

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

\[
N \overset{\lambda_{let,-}}{\longrightarrow} N_0 \overset{\lambda_{let,-}}{\longrightarrow} N'
\]

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 \( N_0 \overset{\lambda_{let,-}}{\longrightarrow} N' \) sequence into \( N_0 \overset{\lambda_{let,-}}{\longrightarrow} N'_0 \overset{\lambda_{let,-}}{\longrightarrow} N' \): The first half of the partition is that required as the trailing let,-C-sequence, and is thus resolved; the inductive step is on \( N'_0 \overset{\lambda_{let,-}}{\longrightarrow} N' \) only. \( \square \)

Lemma A.32. For \( N \overset{\lambda_{let,-}}{\longrightarrow} \),

\[
\begin{array}{c}
M \overset{\lambda_{let,-}}{\longrightarrow} N \\
\lambda_{let} \exists M'
\end{array}
\]

Proof: Any reduction \( M \overset{\lambda_{let}}{\longrightarrow} N \) can be written as

\[
M \overset{\lambda_{let} \setminus \lambda_{let,-}}{\longrightarrow} M_1 \overset{\lambda_{let,-}}{\longrightarrow} M_1,1 \\
\vdots \\
M_n \overset{\lambda_{let} \setminus \lambda_{let,-}}{\longrightarrow} M_n,1 \overset{\lambda_{let,-}}{\longrightarrow} \equiv N.
\]

So the result follows from Lemma A.31 by induction on \( n \). \( \square \)

Theorem A.33, \( \text{SR}(\lambda_{let}) \). \( \lambda_{let,-} \) is a standard reduction relation for \( \lambda_{let} \):

\[
M \overset{\lambda_{let}}{\longrightarrow} A \implies (\exists A') M \overset{\lambda_{let,-}}{\longrightarrow} A'.
\]

Proof: By Lemma A.21 we know that \( A \) is a \( \lambda_{let,-} \)-normal form. So by Lemma A.32 we have \( M' \),

\[
M \overset{\lambda_{let,-}}{\longrightarrow} M' \overset{\lambda_{let} \setminus \lambda_{let,-}}{\longrightarrow} A.
\]

Then by Lemma A.22, \( M' \in A \). \( \square \)
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 → be head reduction in call-by-name λ-calculus and let i denote reduction of a non-head redex, i.e. → ∪ → = →. Furthermore, M →i; N means that we have not only σ : M → N, but also some collection of redexes F contained in M such that σ : (M, F) →cpl N. If → is some reduction relation, we write M →n N to express that M reduces in at most n steps to N.

**Lemma A.34.**

\[
\begin{array}{c}
M \xrightarrow{s} M' \\
\downarrow i, 1 \downarrow i' \\
N \rightarrow s \cdots \rightarrow N'
\end{array}
\]

**Proof:** Take σ : M → M' and i : (M, F) →cpl N. By Lemma 11.4.3(ii) of Barendregt, \( \Delta = \{ \Delta' \} \), where \( \Delta' \) is the head redex of N; take \( N_1 \) such that \( N \rightarrow[i] N_1 \). Also, by Lemma 11.4.3(iii) of Barendregt, every \( \Delta_i \in F \) is an internal redex of N; take \( N_2 \) such that \( (M', F) \rightarrow[i, \sigma] N_2 \). Now we have that both

\[
i' : M \rightarrow[i, i] N \rightarrow \Delta N_1
\]

and

\[
s' : M \rightarrow[i] M' \rightarrow F N_2
\]

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

**Corollary A.35.**

\[
\begin{array}{c}
M \xrightarrow{s} M' \\
\downarrow i \downarrow i' \\
N \rightarrow s \cdots \rightarrow N'
\end{array}
\]

**Proof:** Follows from Lemma A.34 since

\[
M \rightarrow[i] N \Rightarrow (M, \{ \Delta \}) \rightarrow[i] N
\]

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

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

For the inductive step assume $M \xrightarrow{\tau} M' \xrightarrow{\tau}^{n-1} A$ and $M \rightarrow N$. 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 $M \rightarrow N' \xrightarrow{\tau}^{m-1} N$. If $M \rightarrow M'$, it follows that $N' \equiv M'$ since standard reduction is deterministic. By the outer induction hypothesis, $N' \xrightarrow{\tau}^{n-1} A'$, for some answer $A'$. Then, by the inner induction hypothesis, $N \xrightarrow{\tau}^n A'$.

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