Formal Verification of the Correspondence between Call-by-Need and Call-by-Name

Masayuki Mizuno and Eijiro Sumii

Tohoku University

Abstract. We formalize the call-by-need evaluation of non-strict languages and prove its correspondence with call-by-name, using the Coq proof assistant.

It has been long argued that there is a gap between the high-level abstraction of non-strict languages—namely, *call-by-name* evaluation—and their actual *call-by-need* implementations. Although a number of proofs have been given to bridge this gap, they are not necessarily suitable for stringent, mechanized verification because of the use of a global heap, "graph-based" techniques, or "marked reduction". Our technical contributions are twofold: (1) we give a simpler proof based on two forms of standardization, adopting de Bruijn indices for representation of variable bindings along with Ariola and Felleisen's small-step semantics, and (2) we devise a technique to significantly simplify the formalization by eliminating the notion of evaluation contexts—which have been considered essential for the call-by-need calculus—from the definitions.

1 Introduction

Background. The *call-by-name* evaluation strategy has been considered the high-level abstraction of non-strict functional languages since Abramsky [1] and Ong [2] adopted call-by-name evaluation to weak head-normal forms as a formalism of laziness. However, when it comes to actual implementation, call-by-name as it is does not lead to efficient execution because function arguments are evaluated every time they are needed. Therefore, most implementations adopt the *call-by-need* strategy [3], that is: when a redex is found, it is saved in a freshly allocated memory region called a *thunk*; when the redex needs to be evaluated, the thunk is updated with the value of the redex for later reuse.

There has been a large amount of research to bridge the gap between call-by-name and call-by-need by proving their correspondence, that is,

> *if the call-by-need evaluation of a term results in a value, its call-by-name evaluation results in a corresponding value, and vice versa.*

For example, Launchbury [4] defined natural (or "big-step") semantics for call-by-need evaluation and proved its adequacy through denotational semantics. Ariola and Felleisen [5] and Maraist et al. [6] developed small-step call-by-need operational semantics, and proved their correspondence to call-by-name. Kesner [7] gave an alternative proof, based on normalization with non-idempotent intersection types, using Accattoli et al. [8]'s call-by-need semantics.

Existing formalisms and our contribution. In this paper, we mechanize a formalization of the call-by-need evaluation and its correspondence with call-by-name, using the Coq proof assistant.[1] To this goal, after careful design choices, we adapt Ariola and Felleisen's small-step semantics, and give a simpler proof based on two forms of standardization.

In what follows, we review existing formalisms to explain our choices. Several abstract machines (e.g. [9–11]) have been proposed for call-by-need evaluation, but they are generally too low-level for formal verification of correspondence to call-by-name. Launchbury [4] defined call-by-need natural semantics $\Gamma : e \Downarrow \Delta : z$, meaning "term e under store Γ evaluates to value z together with the modified store Δ". Ariola and Felleisen [5] and Maraist et al. [6] independently defined small-step reduction based on let-bindings: $\mathbf{let}\ x = M\ \mathbf{in}\ N$ represents term N with a thunk M pointed to by x. For example, in Ariola and Felleisen's semantics, the term $(\lambda x.xx)((\lambda y.y)(\lambda z.z))$ is reduced as follows:

$$
\begin{aligned}
& (\lambda x.xx)((\lambda y.y)(\lambda z.z)) \\
\to\ & \mathbf{let}\ x = (\lambda y.y)(\lambda z.z)\ \mathbf{in}\ xx && \text{— } x \text{ is bound to } (\lambda y.y)(\lambda z.z) \\
\to\ & \mathbf{let}\ x = (\mathbf{let}\ y = \lambda z.z\ \mathbf{in}\ y)\ \mathbf{in}\ xx && \text{— } x \text{ is evaluated and } y \text{ is bound to } \lambda z.z \\
\to\ & \mathbf{let}\ x = (\underline{\mathbf{let}\ y = \lambda z.z\ \mathbf{in}\ \lambda z.z})\ \mathbf{in}\ xx && \text{— } y \text{ is substituted with its value} \\
\to\ & \underline{\mathbf{let}\ y = \lambda z.z\ \mathbf{in}}\ \mathbf{let}\ x = \lambda z.z\ \mathbf{in}\ xx && \text{— let-bindings are flattened} \\
\to\ & \dots
\end{aligned}
$$

Note, in particular, that the underlined let-binding (of y) is moved forward outside the other let-binding (of x). Such "reassociation" of let-bindings is also required for reductions like $(\mathbf{let}\ x = \dots\ \mathbf{in}\ \lambda y.x)z \to \mathbf{let}\ x = \dots\ \mathbf{in}\ (\lambda y.x)z$.

Variable bindings have been a central topic in the theory of formal languages. Choice of a representation of bindings—such as de Bruijn indices [12], locally nameless representation [13–15], or (parametric) higher order abstract syntax (PHOAS) [16, 17]—is particularly crucial for formal definitions and proofs on proof assistants. We adopt de Bruijn indices along with the reduction in Ariola and Felleisen [5] and Maraist et al. [6] because of their relative simplicity when manipulating the bindings of variables to thunks. (By contrast, Launchbury's semantics is based on a monotonically growing global heap and is still low-level, requiring fresh name generation—a.k.a. "gensym"—for allocation of thunks.) An obstacle in formalizing their definitions is the *evaluation contexts*, which may insert multiple bindings at once and are harder to formalize with de Bruijn indices. We eliminate them by devising a predicate that defines when a term "needs" the value of a variable.

Although our modified *definition* of call-by-need reduction is suitable for formalization with de Bruijn indices, existing *proofs* are still hard to formalize. On one hand, the proof by Ariola and Felleisen [5] is based on informally introduced graph representation of terms for relating call-by-need and call-by-name reductions; as they themselves write, "a graph model for a higher-order language requires many auxiliary notions" [5, p. 3]. On the other hand, Maraist et al. [6]'s proof uses rather intricate "marks" on redexes and their reductions to prove the

[1] We believe that our approach can be adopted in other proof assistants as well.

Syntax

Terms	L, M, N	::=	$x \mid V \mid M\ N$
Values (weak head normal forms)	V	::=	$\lambda x.M$
Evaluation contexts	E_{n}	::=	$[] \mid E_{\mathrm{n}}\ M$

Reduction rule

$$(\beta) \quad (\lambda x.M)N \to M[x \mapsto N]$$

Fig. 1. Definitions for the call-by-name λ-calculus

confluence of their non-deterministic reductions. We therefore devise a simpler proof, outlined as follows. As for the correspondence between terms during call-by-name and call-by-need reductions, we simply inline all let-bindings (denoted by $M^{\Uparrow}$) which represent thunks in call-by-need. Then, roughly speaking, a call-by-need reduction step such as $\mathbf{let}\ x = M\ \mathbf{in}\ N \to \mathbf{let}\ x = M'\ \mathbf{in}\ N$ corresponds to multiple call-by-name steps like $N[x \mapsto M] \to_* N[x \mapsto M']$. We then appeal to Curry and Feys' standardization theorem [18] as follows:

- For the "forward direction", suppose that a term M is evaluated to an answer A by call-by-need. Then $M^{\Uparrow} \xrightarrow{\beta}_* A^{\Uparrow}$ by full β-reduction $\xrightarrow{\beta}$. By a corollary of stadardization, there exists some call-by-name evaluation $M^{\Uparrow} \xrightarrow{\mathrm{name}}_* V$ with $V \xrightarrow{\beta}_* A^{\Uparrow}$.
- The main challenge is to prove the converse direction. Suppose $M^{\Uparrow} \xrightarrow{\mathrm{name}}_* V$. We aim to prove M is evaluated by call-by-need to some answer A corresponding to V. By another corollary of standardization, $M^{\Uparrow}$ is terminating (regardless of non-determinism) by repetition of $\xrightarrow{\mathrm{name}} \circ \xrightarrow{\beta}_*$. Since a call-by-need reduction step corresponds to $\xrightarrow{\mathrm{name}} \circ \xrightarrow{\beta}_*$, the call-by-need evaluation must also terminate with some N. The correspondence between N and V is then proved via the forward direction above.

Our Coq script is available at: `https://github.com/fetburner/call-by-need`

Structure of the paper. We review the call-by-name λ-calculus in Section 2. Section 3 presents the syntax and semantics of Ariola and Felleisen's call-by-need λ-calculus. Section 4 gives the outline of our new proof of the correspondence between call-by-need and call-by-name, based on standardization. Section 5 details our techniques for formalization in a proof assistant (Coq, to be specific). Section 6 discusses previous researches and their relationship to our approach. Section 7 concludes with future work.

2 λ-calculus and call-by-name evaluation

We define the syntax and basic β-reduction of λ-calculus as in Fig. 1. The expression $M[x \mapsto N]$ denotes capture-avoiding substitution of N for each free

occurrence of x in M. The full β-reduction $\xrightarrow{\beta}$ is defined as the compatible closure of the reduction rule (β). For any binary relation $\xrightarrow{R}$, we write the reflexive transitive closure of $\xrightarrow{R}$ as $\xrightarrow{R}_*$. For example, the reflexive transitive closure of $\xrightarrow{\beta}$ is written $\xrightarrow{\beta}_*$.

The call-by-name reduction $\xrightarrow{\text{name}}$ is the closure of the base rule (β) by evaluation contexts E_n. For example, in call-by-name, $(\lambda x.xx)((\lambda y.y)(\lambda z.z))$ is reduced as follows:

$$\begin{array}{l} \quad (\lambda x.xx)((\lambda y.y)(\lambda z.z)) \\ \xrightarrow{\text{name}} (\lambda y.y)(\lambda z.z)((\lambda y.y)(\lambda z.z)) \\ \xrightarrow{\text{name}} (\lambda z.z)((\lambda y.y)(\lambda z.z)) \\ \xrightarrow{\text{name}} (\lambda y.y)(\lambda z.z) \\ \xrightarrow{\text{name}} \lambda z.z \end{array}$$

The relation $\xrightarrow{\text{name}}$ is a partial function and included in $\xrightarrow{\beta}$. Note that, under full β-reduction, there exists a shorter reduction sequence:

$$(\lambda x.xx)((\lambda y.y)(\lambda z.z)) \xrightarrow{\beta} (\lambda x.xx)(\lambda z.z) \xrightarrow{\beta} (\lambda z.z)(\lambda z.z) \xrightarrow{\beta} \lambda z.z$$

In order to establish the correspondence with call-by-need evaluation, we will also focus on stuck states $E_n[x]$. The basic properties of call-by-name can then be summarized as follows:

Lemma 1 (basic properties of call-by-name evaluation).

1. $\xrightarrow{\text{name}}$ *is a partial function.*
2. *If* $E_{\text{n}}[x] = E'_{\text{n}}[y]$ *then* $x = y$.
3. *For any term* M*, exactly one of the following holds:*
   (a) M *is a value*
   (b) $M = E_{\text{n}}[x]$ *for some* E_{n} *and* x
   (c) M *is reducible by* $\xrightarrow{\text{name}}$

Proof. By straightforward structural inductions. □

Thanks to Curry and Feys' standardization theorem, the following Lemma 2 (a call-by-name variant of the leftmost reduction theorem, which is a folklore) and Lemma 3 (a call-by-name variant of quasi-leftmost reduction theorem, which seems to be original) hold. As outlined in the Introduction, they play a crucial role in proving the correspondence between call-by-name and call-by-need.

Lemma 2. *If* $M \xrightarrow{\beta}_* V$*, there exists* V' *such that* $M \xrightarrow{\text{name}}_* V' \xrightarrow{\beta}_* V$.

Lemma 3. *If* $M \xrightarrow{\beta}_* V$ *for some* V*, then* M *is terminating by* $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$ *(despite the non-determinism).*

As we shall see in Section 4, the auxiliary relation $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$ plays a key role when proving the correspondence between call-by-need and call-by-name reductions.

Syntax

Values	V	::=	$\lambda x.M$
Answers	A	::=	$V \mid \mathbf{let}\ x = M\ \mathbf{in}\ A$
Terms	L, M, N	::=	$x \mid V \mid M\ N \mid \mathbf{let}\ x = M\ \mathbf{in}\ N$
Evaluation contexts	E	::=	$[] \mid E\ M \mid \mathbf{let}\ x = M\ \mathbf{in}\ E \mid \mathbf{let}\ x = E\ \mathbf{in}\ E'[x]$

Reduction rules

$$\begin{array}{ll} \text{(I)} & (\lambda x.M)N \to \mathbf{let}\ x = N\ \mathbf{in}\ M \\ \text{(V)} & \mathbf{let}\ x = V\ \mathbf{in}\ E[x] \to \mathbf{let}\ x = V\ \mathbf{in}\ E[V] \\ \text{(C)} & (\mathbf{let}\ x = M\ \mathbf{in}\ A)\ N \to \mathbf{let}\ x = M\ \mathbf{in}\ A\ N \\ \text{(A)} & \mathbf{let}\ y = \mathbf{let}\ x = M\ \mathbf{in}\ A\ \mathbf{in}\ E[y] \to \mathbf{let}\ x = M\ \mathbf{in}\ \mathbf{let}\ y = A\ \mathbf{in}\ E[y] \end{array}$$

Fig. 2. Ariola and Felleisen's call-by-need λ-calculus

3 Ariola and Felleisen's call-by-need λ-calculus

We show the syntax, reduction rules, and evaluation contexts of Ariola and Felleisen's calculus in Fig. 2.[2] By convention, we assume that, whenever we write $E[x]$ on paper, the variable x is not bound by E. Note that our formalization in Coq will use de Bruijn indices and do not need such a convention. The call-by-need reduction $\xrightarrow{\text{need}}$ is the closure of the base rules (I), (V), (C), and (A) by the evaluation contexts E. For example, as mentioned in the introduction, the term $(\lambda x.xx)((\lambda y.y)(\lambda z.z))$ is reduced as follows (the redexes are underlined):

$$\begin{array}{ll} & \underline{(\lambda x.xx)((\lambda y.y)(\lambda z.z))} \\ & \quad \text{— by rule (I) under evaluation context } [] \\ \xrightarrow{\text{need}} & \mathbf{let}\ x = \underline{(\lambda y.y)(\lambda z.z)}\ \mathbf{in}\ xx \\ & \quad \text{— by (I) under } \mathbf{let}\ x = []\ \mathbf{in}\ xx \\ \xrightarrow{\text{need}} & \mathbf{let}\ x = \underline{(\mathbf{let}\ y = \lambda z.z\ \mathbf{in}\ y)}\ \mathbf{in}\ xx \\ & \quad \text{— by (V) under } \mathbf{let}\ x = []\ \mathbf{in}\ xx \\ \xrightarrow{\text{need}} & \underline{\mathbf{let}\ x = (\mathbf{let}\ y = \lambda z.z\ \mathbf{in}\ \lambda z.z)\ \mathbf{in}\ xx} \\ & \quad \text{— by (A) under } [] \\ \xrightarrow{\text{need}} & \mathbf{let}\ y = \lambda z.z\ \mathbf{in}\ \underline{\mathbf{let}\ x = \lambda z.z\ \mathbf{in}\ xx} \\ \xrightarrow{\text{need}} & \ldots \end{array}$$

Note that the value $\lambda z.z$ of x is shared between the two occurrences of x and is never computed twice. Note also that, although the above call-by-need reduction sequence may seem longer than necessary, the "administrative" reductions by (V), (C), and (A) do not contribute to "real" reductions. In order to distinguish administrative reductions when proving the correspondence between call-by-need and call-by-name evaluations, we also consider reductions limited to specific base

[2] Strictly speaking, the reduction rules shown here are called *standard reduction rules* in their paper, as opposed to non-deterministic reduction.

rules as follows. The reduction $\xrightarrow{\text{VCA}}$ is the closure of the three base rules (V), (C), and (A) by evaluation contexts E. Similarly, the reduction $\xrightarrow{\text{I}}$ is defined by the closure of the base rule (I). Obviously, $\xrightarrow{\text{need}} = \xrightarrow{\text{I}} \cup \xrightarrow{\text{VCA}}$.

The points of Ariola and Felleisen's semantics are twofold: the representation of sharing by the syntatic form **let**, and redex positions by evaluation contexts. Thanks to these techniques, their semantics is entirely syntactic, which is desirable for mechanized verification.

The above call-by-need reductions are defined so that they become deterministic:

Lemma 4 (determinacy of call-by-need reductions).

1. $\xrightarrow{\text{I}}$ *is a partial function.*
2. $\xrightarrow{\text{VCA}}$ *is a partial function.*
3. *If* $E[x] = E'[y]$*, then* $x = y$*.*
4. *For any term* M*, exactly one of the following holds:*
   (a) M *is an answer*
   (b) $M = E[x]$ *for some* E *and* x
   (c) M *is reducible by* $\xrightarrow{\text{I}}$
   (d) M *is reducible by* $\xrightarrow{\text{VCA}}$

Proof. Again by straightforward structural inductions. □

4 Outline of our standardization-based proof

Before presenting the Coq formalization, we outline our new proof of the correspondence between call-by-name and call-by-need evaluations, based on standardization.

The correspondence $M^\dagger$ of terms is defined by let expansion as follows:

Definition 1.

$$\begin{aligned} x^\dagger &= x \\ (\lambda x.M)^\dagger &= \lambda x.M^\dagger \\ (M\ N)^\dagger &= M^\dagger\ N^\dagger \\ (\textbf{let}\ x = M\ \textbf{in}\ N)^\dagger &= N^\dagger[x \mapsto M^\dagger] \end{aligned}$$

Although the above definition is similar to Maraist et al. [6], they annotated terms and reductions with what they call "marks", which they use to keep track of the inlined **let**-bindings, while we somehow "recover" their reduction by considering the auxiliary reduction relation $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$.

Lemma 5 (single-step correspondence).

1. $(M[x \mapsto N])^\dagger = M^\dagger[x \mapsto N^\dagger]$.
2. $A^\dagger$ *is a value for any answer* A*.*
3. *For any* E *and* x*,* $E[x]^\dagger = E_{\text{n}}[x]$ *for some* E_{n}*.*

4. *If* $M \xrightarrow{\text{VCA}} N$ *then* $M^{\dagger} = N^{\dagger}$.
5. *If* $M \xrightarrow{\text{I}} N$ *then* $M^{\dagger} \xrightarrow{\text{name}} \circ \xrightarrow{\beta}_{*} N^{\dagger}$.

Note that call-by-name reduction of $M^{\dagger}$ itself does not straightforwardly correspond to call-by-need reduction of M since the latter may reduce more redexes due to the sharing by **let**-bindings. For instance, the call-by-need evaluation

$$\begin{aligned} & \textbf{let } x = (\lambda y.y)(\lambda z.z) \textbf{ in } x(\lambda w.x) \\ \xrightarrow{\text{need}}_{*} & \ldots \textbf{let } x = \lambda z.z \textbf{ in } x(\lambda w.x) \\ \xrightarrow{\text{need}} & \ \ldots \textbf{let } x = \lambda z.z \textbf{ in } (\lambda z.z)(\lambda w.x) \\ \xrightarrow{\text{need}}_{*} & \ldots \textbf{let } x = \lambda z.z \textbf{ in } \ \ldots (\lambda w.x) \end{aligned}$$

(omitting irrelevant **let**-bindings) becomes

$$\begin{aligned} & (\lambda y.y)(\lambda z.z)(\lambda w.(\lambda y.y)(\lambda z.z)) \\ \xrightarrow{\text{name}} & (\lambda z.z)(\lambda w.(\lambda y.y)(\lambda z.z)) \\ \xrightarrow{\text{name}} & \lambda w.(\lambda y.y)(\lambda z.z) \end{aligned}$$

in call-by-name, leaving the β-redex $(\lambda y.y)(\lambda z.z)$ inside a λ-abstraction, which needs to be reduced by full β-reduction.

Proof (Lemma 5). The first two clauses are proved by obvious structural inductions, and the next three clauses follow from the structural inductions on evaluation contexts (note that $\xrightarrow{\text{I}}$ and $\xrightarrow{\text{VCA}}$ are the closure of base rules by evaluation contexts E). □

We first consider the "soundness" direction of the correspondence, that is, any call-by-need evaluation has a corresponding call-by-name evaluation:

Theorem 1 (soundness). *If* $M \xrightarrow{\text{need}}_{*} A$*, then* $M^{\dagger} \xrightarrow{\text{name}}_{*} V \xrightarrow{\beta}_{*} A^{\dagger}$ *for some* V.

Proof. Suppose $M \xrightarrow{\text{need}}_{*} A$. Then $M^{\dagger} \xrightarrow{\beta}_{*} A^{\dagger}$ by clause 4 and 5 of Lemma 5, where $A^{\dagger}$ is a value by clause 2 of Lemma 5. Then, by Lemma 2, we obtain the value V such that $M^{\dagger} \xrightarrow{\text{name}}_{*} V \xrightarrow{\beta}_{*} A^{\dagger}$. □

The harder, converse direction (called "completeness") is as follows:

Theorem 2 (completeness). *If* $M^{\dagger} \xrightarrow{\text{name}}_{*} V$*, then* $M \xrightarrow{\text{need}}_{*} A$ *and* $V \xrightarrow{\beta}_{*} A^{\dagger}$ *for some* A.

In addition to the fact that call-by-name reduction by itself is not "sufficient" for call-by-need as explained above, another problem is that termination under call-by-name does not immediately imply termination under call-by-need since administrative reductions in call-by-need become 0 step in call-by-name, as in clause 4 of Lemma 5. To address the latter issue, we show the termination of administrative reductions as follows:

Lemma 6. *Administrative reduction* $\xrightarrow{\text{VCA}}$ *is terminating.*

Proof. By the decrease of the following measure function $\| M \|_s$, indexed by environments s mapping **let**-bound variables to the measure of their right-hand sides (and defaulting to 1 for other variables).

$$\begin{aligned} \| x \|_s &= s(x) \\ \| \lambda x.M \|_s &= \| M \|_{s \circ [x \mapsto 1]} \\ \| M\ N \|_s &= 2 \| M \|_s + 2 \| N \|_s \\ \| \textbf{let } x = M \textbf{ in } N \|_s &= 2 \| M \|_s + \| N \|_{s \circ [x \mapsto 1 + \| M \|_s]} \end{aligned}$$

□

This proof is simialr to Maraist et al. [6, p. 287] except for our treatment of variables based on environments.

We then prove the completeness theorem:

Proof (Theorem 2). First, we show that the call-by-need reduction of M is normalizing. If there is an infinite call-by-need reduction sequence from M, then by Lemma 6 it must contain an infinite number of $\xrightarrow{\text{I}}$, and therefore by clause 5 of Lemma 5 there is an infinite reduction sequence consisting of $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$ from $M^\dagger$. However, this contradicts with Lemma 3 (since $M^\dagger \xrightarrow{\text{name}}_* V$ obviously implies $M^\dagger \xrightarrow{\beta}_* V$).[3]

Given that M terminates in call-by-need, we next show its normal form N is an answer A. By clause 4 of Lemma 4, if N is *not* an answer, it is *stuck* in call-by-need, that is, $N = E[x]$ for some E and x. Then, by clause 3 of Lemma 5, $E[x]^\dagger = E_n[x]$ for some E_n, that is $N^\dagger$ is stuck in call-by-name. Also, by clause 4 and 5 of Lemma 5, $M^\dagger \xrightarrow{\beta}_* N^\dagger$. Then, by confluence of $\xrightarrow{\beta}$, there exists term L such that $N^\dagger \xrightarrow{\beta}_* L$ and $V \xrightarrow{\beta}_* L$, that is, L is a value and $N^\dagger$ reduces to it by $\xrightarrow{\beta}$. Since stuck states in call-by-name are preserved by $\xrightarrow{\beta}$, this contradicts with the fact that $N^\dagger$ is stuck in call-by-name.

Finally, we show $V \xrightarrow{\beta}_* A^\dagger$. By Theorem 1, we have some V' such that $M^\dagger \xrightarrow{\text{name}}_* V' \xrightarrow{\beta}_* A^\dagger$. We then obtain $V = V'$ by Lemma 1. □

5 Formalization in Coq

The main points of our formalization in Coq are twofold: representation of binding by de Bruijn indices, and implicit treatment of evaluation contexts. We show the syntax and reduction rules of our modified call-by-name and call-by-need λ-calculi in Fig. 3 and Fig. 4.

We use de Bruijn indices for simple manipulation of binding. Fortunately, Ariola and Felleisen's semantics is straightforwardly adaptable for de Bruijn

[3] Although this argument seems to be a proof by contradiction, our actual Coq proof is constructive, using an induction on the finite reduction sequence of $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$ from $M^\dagger$ as we shall see in Section 5.

De Bruijn indexed syntax

$$\begin{array}{lllll} \text{Terms} & L, M, N & ::= & x \mid V \mid M\ N \\ \text{Values} & V & ::= & \lambda.M \end{array}$$

Reduction rules

$$\frac{}{(\lambda x.M)N \xrightarrow{\text{name}} M[x \mapsto N]} \qquad \frac{}{(\lambda x.M)N \xrightarrow{\beta} M[x \mapsto N]}$$

Context rules

$$\frac{}{\mathbf{needs}_{\mathrm{n}}(x, x)} \qquad \frac{\mathbf{needs}_{\mathrm{n}}(M, x)}{\mathbf{needs}_{\mathrm{n}}(M\ N, x)} \qquad \frac{M \xrightarrow{\text{name}} M'}{M\ N \xrightarrow{\text{name}} M'\ N}$$

$$\frac{M \xrightarrow{\beta} M'}{M\ N \xrightarrow{\beta} M'\ N} \qquad \frac{N \xrightarrow{\beta} N'}{M\ N \xrightarrow{\beta} M\ N'} \qquad \frac{M \xrightarrow{\beta} M'}{\lambda.M \xrightarrow{\beta} \lambda.M'}$$

Fig. 3. Our modified definitions for the call-by-name λ-calculus and β-reduction

indices since only a constant number of bindings are inserted or hoisted by a reduction. We use the auxiliary operation $\uparrow_c M$ called "shifting", which increments the indices of the free variables in M above the "cutoff" c as follows:

$$\begin{aligned} \uparrow_c x &= \begin{cases} x & \text{if } x < c \\ x+1 & \text{if } x \geq c \end{cases} \\ \uparrow_c \lambda.M &= \lambda.\uparrow_{c+1} M \\ \uparrow_c (M\ N) &= (\uparrow_c M)\ (\uparrow_c N) \\ \uparrow_c (\mathbf{let}\ \_ = M\ \mathbf{in}\ N) &= (\mathbf{let}\ \_ = \uparrow_c M\ \mathbf{in}\ \uparrow_{c+1} N) \end{aligned}$$

We write $\uparrow M$ for $\uparrow_0 M$. In Coq, we use Autosubst [19] to automatically derive operations such as shifting on terms using de Bruijn indices, and their metatheories including basic properties of substitutions.

Although evaluation contexts reduce the number of reduction rules, explicit treatment of contexts often hinders automated reasoning.[4] For example, consider the clause 4 of Lemma 4. To prove case (b) we need to find a concrete E such that $M = E[x]$, which requires second-order unification [21] in general. More concretely, in Coq, the lemma could be written like

```
Lemma answer_or_stuck_or_reducible M :
  answer M \/
  (exists E x, evalctx E /\ M = E.[tvar x] /\ bv E <= x) \/
```

[4] Another drawback is that evaluation contexts may introduce an arbitrary number of bindings and therefore need to be indexed by that number to coexist with de Bruijn indices, requiring heavy natural number calculations—like the Omega [20] library for Presburger arithmetic—in the mechanized proofs. Our approach will also obviate the need for such calculations.

De Bruijn indexed syntax

Values	V	::=	$\lambda.M$
Answers	A	::=	$V \mid \mathbf{let}\ \_ = M\ \mathbf{in}\ A$
Terms	M, N	::=	$x \mid V \mid M\ N \mid \mathbf{let}\ \_ = M\ \mathbf{in}\ N$

Reduction rules

$$\frac{}{(\lambda.M)\ N \xrightarrow{I} \mathbf{let}\ \_ = N\ \mathbf{in}\ M} \qquad \frac{\mathbf{needs}(M, 0)}{\mathbf{let}\ \_ = V\ \mathbf{in}\ M \xrightarrow{\text{VCA}} M[0 \mapsto V]}$$

$$\frac{}{(\mathbf{let}\ \_ = M\ \mathbf{in}\ A)\ N \xrightarrow{\text{VCA}} \mathbf{let}\ \_ = M\ \mathbf{in}\ A \uparrow N}$$

$$\frac{\mathbf{needs}(N, 0)}{\mathbf{let}\ \_ = (\mathbf{let}\ \_ = M\ \mathbf{in}\ A)\ \mathbf{in}\ N \xrightarrow{\text{VCA}} \mathbf{let}\ \_ = M\ \mathbf{in\ let}\ \_ = A\ \mathbf{in} \uparrow_1 N}$$

Context rules

$$\frac{}{\mathbf{needs}(x, x)} \qquad \frac{\mathbf{needs}(M, x)}{\mathbf{needs}(M\ N, x)} \qquad \frac{\mathbf{needs}(N, x+1)}{\mathbf{needs}(\mathbf{let}\ \_ = M\ \mathbf{in}\ N, x)}$$

$$\frac{\mathbf{needs}(M, x) \qquad \mathbf{needs}(N, 0)}{\mathbf{needs}(\mathbf{let}\ \_ = M\ \mathbf{in}\ N, x)} \qquad \frac{N \xrightarrow{\text{I}} N'}{\mathbf{let}\ \_ = M\ \mathbf{in}\ N \xrightarrow{\text{I}} \mathbf{let}\ \_ = M\ \mathbf{in}\ N'}$$

$$\frac{M \xrightarrow{\text{I}} M'}{M\ N \xrightarrow{\text{I}} M'\ N} \qquad \frac{M \xrightarrow{\text{I}} M' \qquad \mathbf{needs}(N, 0)}{\mathbf{let}\ \_ = M\ \mathbf{in}\ N \xrightarrow{\text{I}} \mathbf{let}\ \_ = M'\ \mathbf{in}\ N}$$

$$\frac{N \xrightarrow{\text{VCA}} N'}{\mathbf{let}\ \_ = M\ \mathbf{in}\ N \xrightarrow{\text{VCA}} \mathbf{let}\ \_ = M\ \mathbf{in}\ N'} \qquad \frac{M \xrightarrow{\text{VCA}} M'}{M\ N \xrightarrow{\text{VCA}} M'\ N}$$

$$\frac{M \xrightarrow{\text{VCA}} M' \qquad \mathbf{needs}(N, 0)}{\mathbf{let}\ \_ = M\ \mathbf{in}\ N \xrightarrow{\text{VCA}} \mathbf{let}\ \_ = M'\ \mathbf{in}\ N}$$

Fig. 4. Our modified call-by-need λ-calculus

```
  (exists E L N, evalctx E /\ M = E.[L] /\ reduceI L N) \/
  (exists E L N, evalctx E /\ M = E.[L] /\ reduceVCA L N).
```

where `bv E <= x` means that x is not captured in E. This statement can be proved by induction on M, where we first encounter the case M is a variable x:

```
4 subgoals

  x : var
  ============================
  answer M \/
  (exists E y, evalctx E /\ tvar x = E.[tvar y] /\ bv E <= y) \/
  (exists E L N, evalctx E /\ tvar x = E.[L] /\ reduceI L N) \/
  (exists E L N, evalctx E /\ tvar x = E.[L] /\ reduceVCA L N)
```

However, automation fails even though the above disjunction is obviously true by the second clause with $E = []$:

```
Coq < eauto.

4 subgoals

  x : var
  ============================
  answer M \/
  (exists E y, evalctx E /\ tvar x = E.[tvar y] /\ bv E <= y) \/
  (exists E L N, evalctx E /\ tvar x = E.[L] /\ reduceI L N) \/
  (exists E L N, evalctx E /\ tvar x = E.[L] /\ reduceVCA L N)
```

We avoid the above problem by eliminating evaluation contexts by expanding their definition in reductions $\xrightarrow{\beta}$, $\xrightarrow{\text{name}}$, $\xrightarrow{\text{I}}$, and $\xrightarrow{\text{VCA}}$, and devising *stuckness predicates* $\mathbf{needs}_{\text{n}}(M, x)$ and $\mathbf{needs}(M, x)$, corresponding to "$M = E_{\text{n}}[x]$ *for some* E_{n}" and "$M = E[x]$ *for some* E", respectively, as in Fig. 4. (Note that, unlike evaluation contexts, each derivation rule of $\mathbf{needs}(M, x)$ inserts only at most one **let**-binding at once; cf. footnote 4.) The only deviation is thunk dereference $\mathbf{let}\ x = V\ \mathbf{in}\ E[x] \xrightarrow{\text{VCA}} \mathbf{let}\ x = V\ \mathbf{in}\ E[V]$, where we approximate the operation $E[V]$ by substitution $(E[x])[x \mapsto V]$. Although the latter may substitute extra occurrences of x in E itself, it is semantically equivalent to the former since V is already a value. Here our formalization favors simplicity over faithfulness and slightly differs from the original definition. It is also straightforward (though cumbersome) to adhere to the original by defining a partial function that substitutes a given value V with a given variable x in a redex position of a given term M.

After the elimination of evaluation contexts, we can now prove clause 4 of Lemma 4 almost automatically as follows:

```
Lemma answer_or_stuck_or_reducible M :
```

```
  answer M \/
  (exists x, needs M x) \/
  (exists N, reduceI M N) \/
  (exists N, reduceVCA M N).
Proof.
  induction M as
    [|? [Hanswer|[[]|[[]|[]]]]
    ||? [Hanswer|[[]|[[]|[]]]] ? [|[[[]]|[[]|[]]]]]; eauto 6;
    inversion Hanswer; subst; eauto 6.
Qed.
```

Let us overview the other changes by our elimination of evaluation contexts: clause 2 of Lemma 1 becomes "*if* $\mathbf{needs}_{\mathrm{n}}(M, x)$ *and* $\mathbf{needs}_{\mathrm{n}}(M, y)$*, then* $x = y$"; case (b) of clause 3 of Lemma 1 changes to "$\mathbf{needs}_{\mathrm{n}}(M, x)$ *for some* x"; clause 3 of Lemma 4 to "*If* $\mathbf{needs}(M, x)$ *and* $\mathbf{needs}(M, y)$*, then* $x = y$"; case (b) of clause 4 of Lemma 4 to "$\mathbf{needs}(M, x)$ *for some* x"; and clause 3 of Lemma 5 to "*If* $\mathbf{needs}(M, x)$ *then* $\mathbf{needs}_{\mathrm{n}}(M^{\pitchfork}, x)$". The proofs of these lemmas proceed by induction on the derivation of $\mathbf{needs}_{\mathrm{n}}(M, x)$ or $\mathbf{needs}(M, x)$ instead of structural induction on evaluation contexts.

Another devisal in our Coq formalization is replacing the *reductio ad impossibilem* for the normalization proof of Theorem 2 (completeness) in Section 4, with an intuitionistic, constructive proof as follows:

Proof (Theorem 2, constructive version). By Lemma 3, $M^{\pitchfork}$ is terminating by $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$. Let us define $L \Downarrow$, meaning that L is terminating by $\xrightarrow{\text{name}} \circ \xrightarrow{\beta}_*$, inductively as: $(\forall L'. L \xrightarrow{\text{name}} \circ \xrightarrow{\beta}_* L' \Rightarrow L' \Downarrow) \Rightarrow L \Downarrow$.[5] We then prove a stronger property that, for any M', $M' \Downarrow$ implies

for any M, if $M' = M^{\pitchfork}$, then M terminates by $\xrightarrow{\text{need}}$

by induction on the definition of $M' \Downarrow$. The above statement is trivially true if M is already a call-by-need normal form. If $M \xrightarrow{\text{I}} N$, then by clause 5 of Lemma 5 we have $M^{\pitchfork} \xrightarrow{\text{name}} \circ \xrightarrow{\beta}_* N^{\pitchfork}$, and by the induction hypothesis we have that N terminates by $\xrightarrow{\text{need}}$. If $M \xrightarrow{\text{VCA}} N$, then by clause 4 of Lemma 5 we have $M^{\pitchfork} = N^{\pitchfork}$ and the conclusion follows from a double, inner induction on reductions by $\xrightarrow{\text{VCA}}$, which is finite by Lemma 6.

The rest of the proof is similar to that in Section 4.

6 Related work

Call-by-name and, to a lesser degree, call-by-need evaluations have been investigated for more than decades. We here focus on notable previous researches on the correspondence between call-by-need and call-by-name (other than Ariola and Felleisen [5], which we have already reviewed in Section 3), and discuss their differences from our approach.

[5] This definition is adopted from the accecibility predicate `Acc` in Coq.

– Launchbury [4] gave a natural semantics for call-by-need evaluation and proved its adequacy with respect to call-by-name evaluation. He defined judgements of the form $\Gamma : e \Downarrow \Delta : z$, meaning "term e under store Γ evaluates to value z, yielding a modified store Δ". The key of his semantics is a "dual use" of variables as pointers to thunks. This technique makes their semantics simpler than conventional operational semantics based on abstract machines.

  However, Launchbury's semantics is still inconvenient for mechanical verification because of subtle variable convention: for example, let us evaluate term $\mathbf{let}\ u = 3, f = (\lambda x.\mathbf{let}\ v = u+1\ \mathbf{in}\ v+x)\ \mathbf{in}\ f\ 2 + f\ 3$. We must rename the bound variable v in the body of the function f every time it called, because the pointers to the thunks are identified with variable names.

– Maraist et al. [6] gave a small-step semantics for call-by-need evaluation and proved its correspondence (full abstraction) with call-by-name. Their semantics is almost the same as Ariola and Felleisen's, making the former's proof method also useful for the latter. (The difference between the two semantics is that, in Maraist et al., variables are values, and an additional reduction rule $\mathbf{let}\ x = M\ \mathbf{in}\ N \to N \quad (x \notin \mathbf{FV}(N))$ is introduced for garbage collection.)

  A point of Maraist et al.'s proof is the introduction of *marks* on redexes for call-by-need reductions (I, V, C, or A). Their approach seems natural for proving the confluence of their non-deterministic reductions but significantly complicates the definitions of terms and reductions in a mechanized metatheory.

  Although we did not directly adopt Maraist et al.'s formalism, it influenced our correspondence $M^{\dagger}$ of call-by-need terms with call-by-name, and our measure function in the proof of Lemma 6.

– Chang and Felleisen [22] proposed a variant of Ariola and Felleisen's semantics and proved its correspondence with Launchbury's semantics. They gave a simpler reduction rule with arguably more complicated evaluation contexts instead of administrative reductions. As argued in Section 5, we avoided any use of evaluation contexts for the sake of easier automation and formalization with de Bruijn indices.

7 Conclusion

We formalized a variant of Ariola and Felleisen's small-step operational semantics of call-by-need λ-calculus and proved its correspondence with call-by-name, using the Coq proof assistant. For the formal verification, we developed a simpler proof based on two forms of standardization (Lemma 2 and 3), adopting de Bruijn indices for representation of variable binding. Along the way, we simplified the formalization and enabled more automation by replacing evaluation contexts with more specific definitions (Section 5).

Future work. We plan to extend our target language to more practical languages. Data types such as tuples and sums should be straightforward since they are already in weak head normal form. The most interesting challenge would be recursive definitions, because they cannot be completely inlined when establishing the correspondence with call-by-name (cf. Definition 1). A natural starting point here may be Ariola and Blom's well-known theory of cyclic λ-calculus [23]. Another (though smaller) issue is binary operations (such as arithmetic addition $+$), for which non-deterministic (but strict) evaluation of the operands may be desirable.

References

1. Abramsky, S.: The Lazy Lambda Calculus. In Turner, D.A., ed.: Research Topics in Functional Programming. Addison-Wesley Publishing Company (1990) 65–116
2. Ong, C.L.: Fully Abstract Models of the Lazy Lambda Calculus. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988, IEEE Computer Society (1988) 368–376
3. Wadsworth, C.P.: Semantics and Pragmatics of the Lambda Calculus. PhD thesis, Oxford University (1971)
4. Launchbury, J.: A Natural Semantics for Lazy Evaluation. In Deusen, M.S.V., Lang, B., eds.: Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993, ACM Press (1993) 144–154
5. Ariola, Z.M., Felleisen, M.: The Call-By-Need lambda Calculus. J. Funct. Program. **7**(3) (1997) 265–301
6. Maraist, J., Odersky, M., Wadler, P.: The Call-by-Need Lambda Calculus. J. Funct. Program. **8**(3) (1998) 275–317
7. Kesner, D.: Reasoning About Call-by-need by Means of Types. In Jacobs, B., Löding, C., eds.: Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Volume 9634 of Lecture Notes in Computer Science., Springer (2016) 424–441
8. Accattoli, B., Barenbaum, P., Mazza, D.: Distilling Abstract Machines. In Jeuring, J., Chakravarty, M.M.T., eds.: Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014, ACM (2014) 363–376
9. Johnsson, T.: Efficient Compilation of Lazy Evaluation. In Deusen, M.S.V., Graham, S.L., eds.: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, Montreal, Canada, June 17-22, 1984, ACM (1984) 58–69
10. Peyton Jones, S.L.: Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. J. Funct. Program. **2**(2) (1992) 127–202
11. Fairbairn, J., Wray, S.: TIM: A simple, lazy abstract machine to execute supercombinatorics. In Kahn, G., ed.: Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings. Volume 274 of Lecture Notes in Computer Science., Springer (1987) 34–45
12. de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae (Proceedings) **75**(5) (1972) 381–392

13. Gordon, A.D.: A Mechanisation of Name-Carrying Syntax up to Alpha-Conversion. In: Higher Order Logic Theorem Proving and its Applications, 6th International Workshop, HUG '93, Vancouver, BC, Canada, August 11-13, 1993, Proceedings. (1993) 413–425
14. McKinna, J., Pollack, R.: Some Lambda Calculus and Type Theory Formalized. J. Autom. Reasoning **23**(3-4) (1999) 373–409
15. McBride, C., McKinna, J.: Functional Pearl: I am not a Number–I am a Free Variable. In Nilsson, H., ed.: Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004, ACM (2004) 1–9
16. Pfenning, F., Elliott, C.: Higher-Order Abstract Syntax. In Wexelblat, R.L., ed.: Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988, ACM (1988) 199–208
17. Chlipala, A.: Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In Hook, J., Thiemann, P., eds.: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008, ACM (2008) 143–156
18. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. Revised edn. Volume 103 of Studies in Logic and the Foundations of Mathematics. North-Holland (1984)
19. Schäfer, S., Tebbi, T., Smolka, G.: Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In Urban, C., Zhang, X., eds.: Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings. Volume 9236 of Lecture Notes in Computer Science., Springer (2015) 359–374
20. Crégut, P.: Omega: a solver of quantifier-free problems in Presburger Arithmetic. In: The Coq proof assistant reference manual. (2017) Version 8.7.0.
21. Goldfarb, W.D.: The Undecidability of the Second-Order Unification Problem. Theor. Comput. Sci. **13** (1981) 225–230
22. Chang, S., Felleisen, M.: The Call-by-Need Lambda Calculus, Revisited. In Seidl, H., ed.: Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Volume 7211 of Lecture Notes in Computer Science., Springer (2012) 128–147
23. Ariola, Z.M., Blom, S.: Cyclic Lambda Calculi. In Abadi, M., Ito, T., eds.: Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings. Volume 1281 of Lecture Notes in Computer Science., Springer (1997) 77–106