A strong call-by-need calculus
Thibaut Balabonski, Antoine Lanco, Guillaume Melquiond

To cite this version:

HAL Id: hal-03149692
https://inria.hal.science/hal-03149692v2
Submitted on 4 May 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
A strong call-by-need calculus

Thibaut Balabonski
Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, Gif-sur-Yvette, 91190, France

Antoine Lanco
Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, Gif-sur-Yvette, 91190, France

Guillaume Melquiond
Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, Gif-sur-Yvette, 91190, France

Abstract

We present a call-by-need λ-calculus that enables strong reduction (that is, reduction inside the body of abstractions) and guarantees that arguments are only evaluated if needed and at most once. This calculus uses explicit substitutions and subsumes the existing strong-call-by-need strategy, but allows for more reduction sequences, and often shorter ones, while preserving the neededness.

The calculus is shown to be normalizing in a strong sense: Whenever a λ-term t admits a normal form n in the λ-calculus, then any reduction sequence from t in the calculus eventually reaches a representative of the normal form n. We also exhibit a restriction of this calculus that has the diamond property and that only performs reduction sequences of minimal length, which makes it systematically better than the existing strategy. We have used the Abella proof assistant to formalize part of this calculus, and discuss how this experiment affected its design.

Introduction

Lambda-calculus is seen as the standard model of computation in functional programming languages, once equipped with an evaluation strategy [26]. The most famous evaluation strategies are call-by-value, which eagerly evaluates the arguments of a function before resolving the function call, call-by-name, where the arguments of a function are evaluated when they are needed, and call-by-need [28, 5], which extends call-by-name with a memoization or sharing mechanism to remember the value of an argument that has already been evaluated.

The strength of call-by-name is that it only evaluates terms whose value is effectively needed, at the (possibly huge) cost of evaluating some terms several times. Conversely, the strength and weakness of call-by-value (by far the most used strategy in actual programming languages) is that it evaluates each function argument exactly once, even when its value is not actually needed and when its evaluation does not terminate. At the cost of memoization, call-by-need combines the benefits of call-by-value and call-by-name, by only evaluating needed arguments and evaluating them only once.

A common point of these strategies is that they are concerned with evaluation, that is computing values. As such they operate in the subset of λ-calculus called weak reduction, in which there is no reduction inside λ-abstractions, the latter being already considered to be values. Some applications however, such as proof assistants or partial evaluation, require reducing inside λ-abstractions, and possibly aiming for the actual normal form of a λ-term.

The first known abstract machine computing the normal form of a term is due to Crégut [16] and implements normal order reduction. More recently, several lines of work have transposed the known evaluation strategies to strong reduction strategies or abstract
machines: call-by-value [19, 10, 3], call-by-name [1], and call-by-need [9, 11]. Some non-advertised strong extensions of call-by-name or call-by-need can also be found in the internals of proof assistants, notably Coq.

These strong strategies are mostly conservative over their underlying weak strategy, and often proceed by iteratively applying a weak strategy to open terms. In other words, they use a restricted form of strong reduction to enable reduction to normal form, but do not try to take advantage of strong reduction to obtain shorter reduction sequences. Since call-by-need has been shown to capture optimal weak reduction [8], it is known that the deliberate use of strong reduction [20] is the only way of allowing shorter reduction sequences.

This paper presents a strong call-by-need calculus, which obeys the following guidelines. First, it only reduces needed redexes. Second, it keeps a level of sharing at least equal to that of call-by-value and call-by-need. Third, it tries to enable strong reduction as generally as possible. This calculus builds on the syntax and a part of the meta-theory of $\lambda$-calculus with explicit substitutions, which we recall in Section 2.

Neededness of a redex is undecidable in general, thus the first and third guidelines are antagonist. Section 3 resolves this tension by exposing a simple syntactic criterion capturing more needed redexes than what is already used in call-by-need strategies. Through reducing only needed redexes, our calculus enjoys a normalization preservation theorem that is stronger than usual: Any $\lambda$-term that is weakly normalizing in the pure $\lambda$-calculus (there is at least one reduction sequence to a normal form, but some other sequences may diverge) will be strongly normalizing in our calculus (any reduction sequence is normalizing). This strong normalization theorem, related to the usual completeness results of call-by-name or call-by-need strategies, is completely dealt with using a system of non-idempotent intersection types. This avoids the traditional tedious syntactic commutation lemmas, hence providing more elegant proofs. This is an improvement over the technique used in previous works [22, 9].

While our calculus contains the strong call-by-need strategy introduced in [9], it also allows more liberal call-by-need strategies that anticipate some strong reduction steps in order to reduce the overall length of the reduction to normal form. Section 4 presents a restriction of the calculus that guarantees reduction sequences of minimal length.

Finally, Section 5 presents a formalization of parts of our results in Abella [6]. Beyond the proof safety provided by such a tool, this formalization has also influenced the design of our strong call-by-need calculus itself in a positive way. In particular, it promoted a presentation based on SOS-style local reduction rules [27], which later became a lever for a more efficient use of non-idempotent intersection types. The formalization can be found at the following address: https://hal.inria.fr/hal-03149692.

2 The host calculus $\lambda_c$

Our strong call-by-need calculus is included in an already known calculus $\lambda_c$, that serves as a technical tool in [9] and which we name our host calculus. This calculus gives the general shape of reduction rules allowing memoization and comes with a system of non-idempotent intersection types. Its reduction however is not constrained by any notion of neededness.

The $\lambda_c$-calculus uses the following syntax of $\lambda$-terms with explicit substitutions, which is isomorphic to the original syntax of the call-by-need calculus using let-bindings [5]:

$$t \in \Lambda_c ::= x | \lambda x.t | t \ t | t[x\ t]$$

The free variables $fv(t)$ of a term $t$ are defined as usual. We call pure $\lambda$-term a term that contains no explicit substitution. We write $C$ for a context, i.e., a term with exactly one
hole □, and \( L \) for a context with the specific shape \( \square[x_1/\!\!t_1]\ldots[x_n/\!\!t_n] \) \((n \geq 0)\). We write \( C[t] \) for the term obtained by plugging the subterm \( t \) in the hole of the context \( C \) (with possible capture of free variables of \( t \) by binders in \( C \)), or \( tL \) when the context is of the specific shape \( L \). We also write \( C[t] \) for plugging a term \( t \) whose free variables are not captured by \( C \).

The values we consider are \( \lambda \)-abstractions.

Reduction in \( \lambda_c \) is defined by the following three reduction rules, applied in any context. Rather than using traditional propagation rules for explicit substitutions [21], it builds on the **Linear Substitution Calculus** [25, 4, 2] which is more similar to the let-in constructs commonly used for defining call-by-need.

\[
\begin{align*}
    (\lambda x.t) u & \rightarrow_{DB} t[x/\!\!u]L \\
    C[x][x/\!\!vL] & \rightarrow_{iv} C[v][x/\!\!vL] \quad \text{with } v \text{ value} \\
    t[x/\!\!u] & \rightarrow_{GE} t \quad \text{with } x \not\in fv(t)
\end{align*}
\]

The rule \( \rightarrow_{DB} \) describes \( \beta \)-reduction “at a distance”. It applies to a \( \beta \)-redex whose \( \lambda \)-abstraction is possibly hidden by a list of explicit substitutions. This rule is a combination of a single use of \( \beta \)-reduction with a repeated use of the structural rule lifting the explicit substitutions at the left of an application. The rule \( \rightarrow_{iv} \) describes the linear substitution of a value, i.e., the substitution of one occurrence of the variable \( x \) bound by an explicit substitution. It has to be understood as a lookup operation. Similarly to \( \rightarrow_{DB} \), this rule embeds a repeated use of a structural rule for unnesting explicit substitutions. Note that this calculus only allows the substitution of \( \lambda \)-abstractions, and not of variables as it is sometimes seen [24]. This restricted behavior is enough for the main results of this paper, and will allow a more compact presentation. Finally, the rule \( \rightarrow_{GE} \) describes garbage collection of an explicit substitution for a variable that does not live anymore. Reduction by any of these rules in any context is written \( t \rightarrow_c u \).

A term \( t \) of \( \lambda_c \) is related to a pure \( \lambda \)-term \( t^* \) by the unfolding operation which applies all the explicit substitutions. We write \( t[x/\!\!u] \) for the meta-level substitution of \( x \) by \( u \) in \( t \).

\[
\begin{align*}
    & x^* = x \\
    & (\lambda x.t)^* = \lambda x.(t^*) \\
    & (tu)^* = t^* u^* \\
    & (t[x/\!\!u]^*) = (t^*)(x/\!\!u^*)
\end{align*}
\]

Through unfolding, any reduction step \( \lambda \rightarrow_c u \) in \( \lambda_c \) is related to a sequence of reductions \( t^* \rightarrow_\beta u^* \) in the pure \( \lambda \)-calculus.

The host calculus \( \lambda_c \) comes with a system of non-idempotent intersection types [15, 18], defined in [23] by adding explicit substitutions to an original system from [18]. A type \( \tau \) may be a type variable \( \alpha \) or an arrow type \( M \rightarrow \tau \), where \( M \) is a multiset \( \{\sigma_1, \ldots, \sigma_n\} \) of types. A typing environment \( \Gamma \) associates to each variable in its domain a multiset of types. This multiset contains one type for each potential use of the variable, and may be empty if the variable is not actually used. A typing judgment \( \Gamma \vdash t : \tau \) assigns exactly one type to the term \( t \). As shown by the typing rules in Fig. 1, an argument of an application or of an explicit substitution may be typed several times in a derivation. Note that, in the rules, the subscript \( \sigma \in M \) quantifies on all the instances of elements in the multiset \( M \). This type system is known to characterize \( \lambda \)-terms that are weakly normalizing for \( \beta \)-reduction, if associated with the side condition that the empty multiset \( \{\} \) does not appear at a positive position in the typing judgment. Positive type occurrences \( T_+(\Gamma \vdash t : \tau) \) and negative type occurrences \( T_-(\Gamma \vdash t : \tau) \) of a typing judgment are defined by the following equations.

\[
\begin{align*}
    T_+(\alpha) &= \{\alpha\} \\
    T_+(M) &= \{M\} \cup \bigcup_{\sigma \in M} T_+(\sigma) \\
    T_+(M \rightarrow \sigma) &= \{M \rightarrow \sigma\} \cup T_-(M) \cup T_+(\sigma) \\
    T_+(\Gamma \vdash t : \sigma) &= T_+(\sigma) \cup \bigcup_{x \in \text{dom}(\Gamma)} T_-(\Gamma(x)) \\
    T_-(\alpha) &= \emptyset \\
    T_-(M) &= \bigcup_{\sigma \in M} T_-(\sigma) \\
    T_-(M \rightarrow \sigma) &= T_+(M) \cup T_-(\sigma) \\
    T_-(\Gamma \vdash t : \sigma) &= T_+(\Gamma) \cup T_-(\Gamma(x))
\end{align*}
\]
A strong call-by-need calculus

TY-VAR
\[ x : \sigma \vdash x : \sigma \]

TY-@ 
\[ \Gamma \vdash t : M \rightarrow \tau \quad (\Delta_{\sigma} \vdash u : \sigma)_{\sigma \in M} \]
\[ \Gamma + \sum_{\sigma \in M} \Delta_{\sigma} \vdash t \ u : \tau \]

TY-λ 
\[ \Gamma; x : M \vdash t : \tau \] 
\[ \Gamma \vdash \lambda x.t : M \rightarrow \tau \]

TY-ES 
\[ \Gamma; x : M \vdash t : \tau \quad (\Delta_{\sigma} \vdash u : \sigma)_{\sigma \in M} \]
\[ \Gamma + \sum_{\sigma \in M} \Delta_{\sigma} \vdash t[x/u] : \tau \]

Figure 1 Typing rules for \( \lambda_c \).

Theorem 1 (Typability [17, 12]). If the pure \( \lambda \)-term \( t \) is weakly normalizing for \( \beta \)-reduction, then there is a typing judgment \( \Gamma \vdash t : \tau \) such that \( \{\} \notin T_+(\Gamma \vdash t : \tau) \).

A typing derivation \( \Phi \) for a typing judgment \( \Gamma \vdash t : \tau \) (written \( \Phi \triangleright \Gamma \vdash t : \tau \)) defines in \( t \) a set of typed positions, which are the positions of the subterms of \( t \) for which the derivation \( \Phi \) contains a subderivation. More precisely:
- \( \varepsilon \) is a typed position for any derivation;
- if \( \Phi \) ends with rule TY-λ, TY-@ or TY-ES, then 0p is a typed position of \( \Phi \) if \( p \) is a typed position of the subderivation \( \Phi' \) relative to the first premise;
- if \( \Phi \) ends with rule TY-@ or TY-ES, then 1p is a typed position of \( \Phi \) if \( p \) is a typed position of the subderivation \( \Phi' \) relative to one of the instances of the second premise.

Note that, in the latter case, there is no instance of the second premise and no typed position 1p when the multiset \( M \) is empty. On the contrary, when \( M \) has several elements, we get the union of the typed positions contributed by each instance.

These typed positions have an important property; they satisfy a weighted subject reduction theorem ensuring a decreasing derivation size, which we will use in the next section. We call size of a derivation \( \Phi \) the number of nodes of the derivation tree.

Theorem 2 (Weighted subject reduction [9]). If \( \Phi \triangleright \Gamma \vdash t : \tau \) and \( t \rightarrow c t' \) by reduction of a redex at a typed position, then there is a derivation \( \Phi' \triangleright \Gamma \vdash t' : \tau \) with \( \Phi' \) smaller than \( \Phi \).

3 Strong call-by-need calculus \( \lambda_{sn} \)

Our strong call-by-need calculus is defined by the same terms and reduction rules as \( \lambda_c \), with restrictions on where the reduction rules can be applied. These restrictions ensure in particular that only the needed redexes are reduced. Notice that gc-reduction is never needed in this calculus and will thus be ignored from now on.

3.1 Reduction in \( \lambda_{sn} \)

The main reduction relation is written \( t \rightarrow_{sn} t' \) and represents one step of dB or lsv reduction at an eligible position of the term \( t \). The starting point is the same as the one for the original (weak) call-by-need calculus. Since the argument of a function is not always needed, we do not reduce in advance the right part of an application \( tu \). Instead, we first evaluate \( t \) to an answer \( (\lambda x.t')L \), then apply a dB-reduction to put the argument \( u \) in the environment of \( t' \), and then go on with the resulting term \( t'[x/u]L \), evaluating \( u \) only if and when it is required.
Strong reduction. The previous principle is enough for weak reduction, but new behaviors appear with strong reduction. For instance, consider the top-level term $\lambda x.x \ t \ u$. It is an abstraction, which would not need to be further evaluated in weak call-by-need. Here however, we have to reduce it further to reach its putative normal form. So, let us gather some knowledge on the term. Given its position, we know that this abstraction will never be applied to an argument. This means in particular that its variable $x$ will never be substituted by anything; it is blocked and is now part of the rigid structure of the term. Following [9], we say that variable $x$ is frozen. As for the arguments $t$ and $u$ given to the frozen variable $x$, they will always remain at their respective positions and their neededness is guaranteed. So, the calculus allows their reduction. Moreover, these subterms $t$ and $u$ will never be applied to other subterms; they are in top-level-like positions and can be treated as independent terms. In particular, assuming that the top-level term is $\lambda x. (\lambda y.t') u$ (that is, $t$ is the abstraction $\lambda y.t'$), the variable $y$ will never be substituted and both variables $x$ and $y$ can be seen as frozen in the subterm $t'$.

Let us now consider the top-level term $(\lambda x. (\lambda y.t') u) v$, i.e., the previous one applied to some argument $v$. The analysis becomes radically different. Indeed, both abstractions in this term are at positions where they may eventually interact with other parts of the term: $(\lambda x \ldots)$ is already applied to an argument, while $(\lambda y.t')$ might eventually be substituted at some position inside $v$ whose properties are not yet known. Thus, none of the abstractions is at a top-level-like position and we cannot rule out the possibility that some occurrences of $x$ or $y$ become substituted eventually. Consequently, neither $x$ nor $y$ can be considered as frozen. In addition, notice that the subterms $\lambda y.t'$ and $u$ are not even guaranteed to be needed in $(\lambda x. (\lambda y.t') u) v$. Thus our calculus shall not allow them to be reduced yet.

Therefore, the set of top-level-like positions of a subterm $t$, and more importantly the set of its positions that are eligible for reduction largely depend on the context surrounding $t$. Consequently, the bulk of the definition of $t \rightarrow_{sn} t'$ is an inductive relation $t \overset{\mu,\varphi}{\rightarrow}_{sn} t'$ that plays two roles: identifying a position where a reduction rule can be applied in $t$, depending on some outer context information, and performing said reduction. The information on the context is abstracted by two parameters of the inductive relation:

- a flag $\mu$ indicating whether $t$ is at a top-level-like position ($\top$) or not ($\bot$);
- the set $\varphi$ of variables that are frozen at the considered position.

The flow of this information along the inductive rules is a critical aspect of the definitions.

Since the identification of positions that are eligible for reduction does not depend on the choice of the rule dB or lsv, the inductive reduction relation is also parametric with respect to the rule. This is the role of the parameter $\rho$ of $\overset{\rho,\varphi,\mu}{\rightarrow}_{sn}$, whose value can be dB, lsv, or others that we will introduce shortly. Thus, the top-level reduction relation $t \rightarrow_{sn} t'$ holds whenever $\vdash_{\rho,\varphi,\mu,\top} t \rightarrow_{sn} t'$ or $\vdash_{\rho,\varphi,\mu,\bot} t \rightarrow_{sn} t'$, where the flag $\mu$ is $\top$, and the set $\varphi$ is typically empty when $t$ is closed, or contains the free variables of $t$ otherwise.

Inductive rules. The inference rules for $\overset{\rho,\varphi,\mu}{\rightarrow}_{sn}$ are given in Fig. 3. Notice that information about $\varphi$ and $\mu$ flow outside-in, that is from top-level to the position of the reduction, or equivalently upward in the inference rules, while $\rho$ flows the other way. Notice also that in this paper, we define top-level-like positions and frozen variables only through these inductive rules.

Rule @-LEFT makes reduction always possible on the left of an application, but as shown by the premise this position is not a $\top$ position. Rule @-RIGHT on the other hand allows reducing on the right of an application, and even doing so in $\top$ mode, but only when the application is led by a frozen variable.
A strong call-by-need calculus

As implied by the last rule in Fig. 2, an explicit substitution in a structure may even affect the set of frozen variables. It differs from the notion of structure, which is an application \( x \ t_1 \ldots t_n \) led by a frozen variable \( x \), possibly interlaced with explicit substitutions (Fig. 2). As implied by the last rule in Fig. 2, an explicit substitution in a structure may even affect the leading variable, provided that the content of the substitution is itself a structure. We write \( S_\varphi \) the set of structures under a set \( \varphi \) of frozen variables. It differs from the notion in [9] in that it does not require the term to be in normal form.

Notice that frozen variables in a term \( t \) are either free variables of \( t \), or variables introduced by binders in \( t \). As such they obey the usual renaming conventions. In particular, the third and fourth rules in Fig. 2 implicitly require that the variable \( x \) bound by the explicit substitution is fresh and hence not in the set \( \varphi \). We keep this freshness convention in all the definitions of the paper.

Rules \( \lambda\text{-TOP} \) and \( \lambda\text{-BOT} \) make reduction always possible inside a \( \lambda \)-abstraction, i.e., unconditional strong reduction. If the abstraction is in a \( \top \) position, its variable is added to the set of frozen variables while reducing the body of the abstraction. Rules \( \text{ES-LEFT}\text{-}\varphi \) and \( \text{ES-LEFT-}\varphi \) show that it is always possible to reduce a term affected by an explicit substitution. If the substitution contains a structure, the variable bound by the substitution can be added to the set of frozen variables. Rule \( \text{ES-RIGHT} \) restricts reduction inside a substitution to the case where an occurrence of the substituted variable is at a reducible position. It uses an auxiliary rule \( \text{id}_x \), which propagates using the same inductive reduction relation, to probe a term for the presence of some variable \( x \) at a reduction position. By freshness, \( x \notin \varphi \). This auxiliary rule does not modify the term to which it applies, as witnessed by its base case \( \text{id} \).

Rules \( \text{dB} \) and \( \text{LSV} \) are the base cases for applying reductions \( \text{dB} \) or \( \text{LSV} \). Using the notations of \( \ell_c \), they allow the following reductions.

\[
\begin{align*}
(x \in \varphi & \quad t \in S_\varphi & \quad t \in S_\varphi & \quad t \in S_\varphi \cup \{x\} & \quad u \in S_\varphi \\
x \in S_\varphi & \quad t \in S_\varphi & \quad t[x \backslash u] \in S_\varphi & \quad t[x \backslash u] \in S_\varphi
\end{align*}
\]
Each is defined using an auxiliary reduction relation dealing with the list \( L \) of explicit substitutions. These auxiliary reductions are given in Fig. 4.

Rules \( \text{DB-BASE} \) and \( \text{LSV-BASE} \) describe the base cases of the auxiliary reductions, where the list \( L \) is empty. Note that, while \( \text{DB-BASE} \) is an axiom, the inference rule \( \text{LSV-BASE} \) uses as a premise a reduction \( \rho \vdash \phi \vdash_{\text{rb}} \) using a new reduction rule \( \rho = \text{sub}_v \). This reduction rule substitutes one occurrence of the variable \( x \) at a reducible position by the value \( v \) (with, by freshness, \( x \not\in \phi \)). As seen for \( \text{id}_x \) above, this reduction rule propagates using the same inductive reduction relation, and its base case is the rule \( \text{sub} \) in Fig. 3. The presence of this premise \( t \vdash \text{sub}_v \vdash \phi \vdash_{\text{rb}} t' \) in the rule is the primary reason why the auxiliary relation \( \phi \vdash_{\text{rb}} \) is parameterized by \( \phi \) and \( \mu \). The combination of the rules \( \text{LSV} \) and \( \text{LSV-BASE} \) makes it possible, in the case of a \( \text{LSV} \)-reduction, to resume the search for a reducible variable in the context in which the substitution has been found (instead of resetting the context). In [9], a similar effect was achieved using a more convoluted condition on a composition of contexts.

Rule \( \text{DB-}\) makes it possible to float out an explicit substitution applied to the left part of an application. That is, if a \( \text{DB} \)-reduction is possible without the substitution, then the reduction is performed and the substitution is applied to the result. Rules \( \text{LSV-}\) and \( \text{LSV-}\) achieve the same effect with the nested substitutions applied to the value substituted by an \( \text{LSV} \)-reduction step. As with rule \( \text{ES-LEFT}\), if the substitution is a structure, the variable can be frozen. This difference between \( \text{LSV-}\) and \( \text{LSV-}\) can be ignored until Sec. 4.

Finally, note that the strong call-by-need strategy introduced in [9] is included in our calculus. One can recover this strategy by imposing two restrictions on \( \rho \vdash \phi \vdash_{\text{rb}} \):

- remove the rule \( \lambda\)-\( \text{BOT} \), so as to only reduce abstractions that are in top-level-like positions;
- restrict the rule \( \text{DIR}\) to the case where the left member of the application is a structure in normal form, since the strategy imposes left-to-right reduction of structures.

\textbf{Example.} The reduction \( (\lambda a.a \, x)[x/(\lambda y.t)[z/u]] \rightarrow^{\text{sn}} (\lambda a.a \, x)[x'/t[y/v]/[z/u]] \) is allowed by \( \lambda_{\text{sn}} \), as shown by the following derivation. The left branch of the derivation checks that an occurrence of the variable \( x \) is actually at a needed position in \( \lambda a.a \, x \), while its right branch reduces the argument of the substitution.
3.2 Soundness

The calculus $\lambda_{sn}$ is sound with respect to the $\lambda$-calculus, in the sense that any normalizing reduction in $\lambda_{sn}$ can be related to a normalizing $\beta$-reduction through unfolding. This section establishes this result (Th. 6). All proofs in this section are formalized in Abella.

The first part of the proof requires relating $\lambda_{sn}$-reduction to $\beta$-reduction.

▶ Lemma 3 (Simulation). If $t \rightarrow_{\lambda_{sn}} t'$ then $t^* \rightarrow_{\beta} t'^*$.

Proof. By induction on the reduction $t \rightarrow_{\lambda_{sn}} t'$.

The second part requires relating the normal forms of $\lambda_{sn}$ to $\beta$-normal forms. The normal forms of $\lambda_{sn}$ correspond to the normal forms of the strong call-by-need strategy [9]. They can be characterized by the inductive definition given in Fig. 5.

▶ Lemma 4 (Normal forms). $t \in N_{\phi}$ if and only if there is no reduction $t \rightarrow_{\lambda_{sn}} t'$.

Proof. The first part (a term cannot be in normal form and reducible) is by induction on the reduction rules. The second part (any term is either a normal form or a reducible term) is by induction on $t$.

▶ Lemma 5 (Unfolded normal forms). If $t \in N_{\phi}$ then $t^*$ is a normal form in the $\lambda$-calculus.

Proof. By induction on $t \in N_{\phi}$.

Soundness is a direct consequence of the three previous lemmas.

▶ Theorem 6 (Soundness). Let $t$ be a $\lambda_{sn}$-term. If there is a reduction $t \rightarrow_{\lambda_{sn}} u$ with $u$ a $\lambda_{sn}$-normal form, then $u^*$ is the $\beta$-normal form of $t^*$.

This theorem implies that all the $\lambda_{sn}$-normal forms of a term $t$ are equivalent modulo unfolding. This mitigates the fact that the calculus, without a gc rule, is not confluent. For instance, the term $(\lambda x.x) (\lambda y. (\lambda z.z) y)$ admits two normal forms $(\lambda y. z [z \backslash x]) [x \backslash y] \lambda y. (\lambda z.z) y$ and $(\lambda y. z [z \backslash y]) [x \backslash y] \lambda y. z [z \backslash y]$, but both of them unfold to $\lambda y. y$.

3.3 Completeness

Our strong call-by-need calculus is complete with respect to normalization in the $\lambda$-calculus in a strong sense: Whenever a $\lambda$-term $t$ admits a normal form in the pure $\lambda$-calculus, every reduction path in $\lambda_{sn}$ eventually reaches a representative of this normal form. This section is devoted to proving this completeness result (Th. 12). The proof relies on the non-idempotent intersection type system in the following way. First, typability (Th. 1) ensures that any weakly normalizing $\lambda$-term admits a typing derivation (with no positive occurrence of $\emptyset$).
Second, we prove that any \(\lambda_{\mathfrak{n}}\)-reduction in a typed \(\lambda_{\mathfrak{n}}\)-term \(t\) (with no positive occurrence of \(\emptyset\)) is at a typed position of \(t\) (Th. 11). Third, weighted subject reduction (Th. 2) provides a decreasing measure for \(\lambda_{\mathfrak{n}}\)-reduction. Finally, the obtained normal form is related to the \(\beta\)-normal form using Lemmas 3, 4, and 5.

The proof of the forthcoming typed reduction (Th. 11) uses a refinement of the non-idempotent intersection types system of \(\lambda_{\mathfrak{c}}\), given in Fig. 6. Both systems derive the same typing judgments with the same typed positions. The refined system however features an annotated typing judgment \(\Gamma \vdash^\mu_{\emptyset} t : \tau\) embedding the same context information that are used in the inductive reduction relation \(\rho_{\emptyset,\mu,\mathfrak{n}}\), namely the set \(\varphi\) of frozen variables at the considered position and a marker \(\mu\) of top-level-like positions. These annotations are faithful counterparts to the corresponding annotations of \(\lambda_{\mathfrak{n}}\) reduction rules; their information flows upward in the inference rules following the same criteria.

In particular, the rule for typing an abstraction is split in two versions \(\text{TY-}\lambda\rightarrow\perp\) and \(\text{TY-}\lambda\rightarrow\top\), the latter being applicable to \(\top\) positions and thus freezing the variable bound by the abstraction (in both rules, by freshness convention we assume \(x \notin \varphi\)). The rule for typing an application is also split in two version: \(\text{TY-}\emptyset\rightarrow\top\) is applicable when the left part of the application is a structure and marks the right part as a \(\top\) position, while \(\text{TY-}\emptyset\rightarrow\perp\) is applicable otherwise. Note that this second rule allows the argument of the application to be typed even if its position is not (yet) reducible, but its typing is in a \(\perp\) position. Finally, the rule for typing an explicit substitution is similarly split in two versions, depending on whether the content of the substitution is a structure or not, and handling the set of frozen variables accordingly. In both cases, the content of the substitution is typed in a \(\perp\) position, since this position is never top-level-like. We write \(\Phi \triangleright \Gamma \vdash^\mu_{\emptyset} t : \tau\) if there is a derivation \(\Phi\) of the annotated typing judgment \(\Gamma \vdash^\mu_{\emptyset} t : \tau\). We denote \(\mathsf{fzt}(\Phi)\) the set of types associated to frozen variables in judgments of the derivation \(\Phi\).

\section*{Lemma 7 (Typing derivation annotation).} If there is a derivation \(\Phi \triangleright \Gamma \vdash t : \tau\), then for any \(\varphi\) and \(\mu\) there is a derivation \(\Phi' \triangleright \Gamma \vdash^\mu_{\emptyset} t : \tau\) such that the sets of typed positions in \(\Phi\) and \(\Phi'\) are equal.

\section*{Proof.} By induction on \(\Phi\), since annotations do not interfere with typing. \(\blacksquare\)

The converse property is also true, by erasing of the annotations, but is not used in the proof of the completeness result.
A strong call-by-need calculus

The most crucial part of the proof of Th. 11 is ensuring that any argument of a typed structure is itself at a typed position. This follows from the following three lemmas.

**Lemma 8** (Typed structure). If $\Gamma \vdash^\rho \mu t : \tau$ and $t \in S_\varphi$, there is $x \in \varphi$ such that $\tau \in T_\tau(\Gamma(x))$.

**Proof.** By induction on the structure of $t$. The most interesting case is the one of an explicit substitution $t_1[x]\backslash t_2$. The induction hypothesis applied on $t_1$ can give the variable $x$ which does not appear in the conclusion, but in that case $t_2$ is guaranteed to be a structure whose type contains $\tau$.

**Lemma 9** (Subformula property).

1. If $\Phi \vdash \Gamma \vdash^\rho t : \tau$ then $\bigcup_{x \in \varphi} T_\tau(\Gamma(x)) \cup T_\tau(\tau)$
2. If $\Phi \vdash \Gamma \vdash^\rho t : \tau$ then $\bigcup_{x \in \varphi} T_\tau(\Gamma(x))$

**Proof.** By mutual induction on the typing derivations. Most cases are fairly straightforward. The only difficult case comes from the rule $\textit{ty-@-S}$, in which there is a premise $\Delta \vdash^\rho \mu u : \sigma$ with mode $\top$ but with a type $\sigma$ that does not clearly appear in the conclusion. Here we need the typed structure (Lem. 8) to conclude.

**Lemma 10** (Typed structure argument). If $\Phi \vdash \Gamma \vdash^\rho t : \tau$ with $\not\in T_\tau(\Gamma \vdash t : \tau)$, then every typing judgment of the shape $\Gamma' \vdash^\rho s : \mathcal{M} \rightarrow \sigma$ in $\Phi$ with $s \in S_\varphi$ satisfies $\mathcal{M} \neq \not\in$.

**Proof.** Let $\Gamma' \vdash^\rho_\varphi s : \mathcal{M} \rightarrow \sigma$ in $\Phi$ with $s \in S_\varphi$. By Lemma 8, there is $x \in \varphi'$ such that $\mathcal{M} \rightarrow \sigma \in T_\tau(\Gamma'(x))$. Then $\mathcal{M} \in T_\tau(\Gamma'(x))$ and $\mathcal{M} \in T_\tau(fzt(\Phi))$. By Lemma 9, $\mathcal{M} \in T_\tau(\Gamma \vdash^\rho t : \tau)$, thus $\mathcal{M} \neq \not\in$.

**Theorem 11** (Typed reduction). If $\Phi \vdash \Gamma \vdash^\rho t : \tau$ with $\not\in T_\tau(\Gamma \vdash t : \tau)$, then every $\lambda_n$-reduction $t \overset{\rho_\varphi, \mu_\tau}{\rightarrow}_{\gamma_n} t'$ is at a typed position.

**Proof.** We prove by induction on $t \overset{\rho_\varphi, \mu_\tau}{\rightarrow}_{\gamma_n} t'$ that, if $\Phi \vdash \Gamma \vdash^\rho t : \tau$ with $\Phi$ such that any typing judgment $\Gamma' \vdash^\rho t : \tau$ in $\Phi$ with $s \in S_\varphi$ satisfies $\mathcal{M} \neq \not\in$, then $t \overset{\rho_\varphi, \mu_\tau}{\rightarrow}_{\gamma_n} t'$ reduces at a typed position (the restriction on $\Phi$ is enabled by Lemma 10). Since all other reduction cases concern positions that are systematically typed, we focus here on $\textit{@-RIGHT}$ and $\textit{es-right}$.

- Case $\textit{@-RIGHT}$: $t u \overset{\rho_\varphi, \mu_\tau}{\rightarrow}_{\gamma_n} t' u'$ with $t \in S_\varphi$ and $u \overset{\rho_\varphi, \mu_\tau}{\rightarrow}_{\gamma_n} u'$, assuming $\Phi \vdash \Gamma \vdash^\rho t u : \sigma$. By inversion of the last rule in $\Phi$ we know there is a subderivation $\Phi' \vdash \Gamma \vdash^\rho t : \mathcal{M} \rightarrow \sigma$ and by hypothesis $\mathcal{M} \neq \not\in$. Then $u$ is typed in $\Phi$ and we can conclude by induction hypothesis.

- Case $\textit{es-right}$: $t[x\backslash u] \overset{\rho_\varphi, \mu_\tau}{\rightarrow}_{\gamma_n} t[x\backslash u']$ with $t \overset{id_\varphi, \varphi_\tau}{\rightarrow}_{\gamma_n} t'$ and $u \overset{\rho_\varphi, \mu_\tau}{\rightarrow}_{\gamma_n} u'$, assuming $\Phi \vdash \Gamma \vdash^\rho t[x\backslash u] : \tau$. By inversion of the last rule in $\Phi$ we kown there is a subderivation $\Phi' \vdash \Gamma' : \mathcal{M} \vdash^\rho t : \tau$. By induction hypothesis we know that reduction $t \overset{id_\varphi, \varphi_\tau}{\rightarrow}_{\gamma_n} t'$ is at a typed position in $\Phi'$, thus $x$ is typed in $t$ and $\mathcal{M} \neq \not\in$. Then $u$ is typed in $\Phi$ and we can conclude by induction hypothesis on $u \overset{\rho_\varphi, \mu_\tau}{\rightarrow}_{\gamma_n} u'$.

**Theorem 12** (Completeness). If a $\lambda$-term $t$ is weakly normalizing in the $\lambda$-calculus, then $t$ is strongly normalizing in $\lambda_n$. Moreover, if $n_\beta$ is the normal form of $t$ in the $\lambda$-calculus, then any normal form $n_{sn}$ of $t$ in $\lambda_n$ is such that $n_{sn}^* = n_\beta$.

---

1 See appendix for the complete proof.
Figure 7 New rule LSV-BASE for $\lambda_{sn^+}$.

Proof. Let $t$ be a pure $\lambda$-term that admits a normal form $n_{\beta}$ for $\beta$-reduction. By Theorem 1 there exists a derivable typing judgment $\Gamma \vdash t : \tau$ such that $\emptyset \notin T_+(\Gamma \vdash t : \tau)$. Thus by Theorems 11 and 2, the term $t$ is strongly normalizing for $\rightarrow_{sn}$. Let $t \rightarrow_{sn}^* n_{sn}$ be a maximal reduction in $\lambda_{sn}$. By Lemma 4, $n_{sn} \in N_{\phi}$, and by Lemma 5, $n_{sn}$ is a normal form in the $\lambda$-calculus. Moreover, by simulation (Lem. 3), there is a reduction $t^* \rightarrow_{\beta}^* n_{\beta}$. By uniqueness of the normal form in the $\lambda$-calculus, $n_{\beta} = n_{sn}$. ◀

Note that, despite the fact that $\lambda_{sn}$ does not enjoy the diamond property, our theorems of soundness (Th. 6) and completeness (Th. 12) imply that, in $\lambda_{sn}$, a term is weakly normalizing if and only if it is strongly normalizing.

4 Relatively optimal strategies

Our proposed $\lambda_{sn}$-calculus guarantees that, in the process of reducing a term to its strong normal form, only needed redexes are ever reduced. This does not tell anything about the length of reduction sequences, though. Indeed, a term might be substituted several times before being reduced, thus leading to duplicate computations. To prevent this duplication, we introduce a notion of local normal form, which is used to restrict the value criterion in the LSV-BASE rule. This restricted calculus, named $\lambda_{sn^+}$, has the same rules as $\lambda_{sn}$ (Fig. 3 and 4), except that LSV-BASE is replaced by the rule shown in Fig. 7.

We then show that this restriction is strong enough to guarantee the diamond property. Finally, we explain why our restricted calculus only produces minimal sequences, among all the reduction sequences allowed by $\lambda_{sn}$. This makes it a relatively optimal strategy.

4.1 Local normal forms

In $\lambda_c$ and $\lambda_{sn}$, substituted terms can be arbitrary values. In particular, they might be abstractions whose body contains some redexes. Since substituted variables can appear multiple times, this would cause the redex to be reduced several times if the value is substituted too soon. Let us illustrate this phenomenon on the following example, where $id = \lambda x.x$. The sequence of reductions does not depend on the set $\phi$ of frozen variables nor on the position $\mu$, so we do not write them to lighten a bit the notations. Subterms that are about to be substituted or reduced are underlined.

\[
\begin{align*}
(\lambda w. w) (\lambda y. y) & \xrightarrow{db}_{\lambda_{sn}} (w \ w)[w \\lambda y. id \ y] \\
& \xrightarrow{db}_{\lambda_{sn}} ((\lambda y. id \ y) \ w)[w \\lambda y. id \ y] \\
& \xrightarrow{db}_{\lambda_{sn}} ((\lambda y. x[\ y\ ] ) \ w)[w \\lambda y. id \ y] \\
& \xrightarrow{db}_{\lambda_{sn}} \emptyset [\ y\ ] [w \\lambda y. id \ y] \\
& \xrightarrow{db}_{\lambda_{sn}} (\lambda y. id \ y)[x \\lambda y. id \ y][y \\lambda y. id \ y][w \\lambda y. id \ y] \\
& \xrightarrow{db}_{\lambda_{sn}} (\lambda y. x[\ y\ ] ) [x \\lambda y. id \ y][y \\lambda y. id \ y][w \\lambda y. id \ y] 
\end{align*}
\]
**A strong call-by-need calculus**

\[x \in \varphi \cup \omega\]

\[x \in N_{\varphi,\omega,\mu}\]

\[t \in N_{\varphi,\omega,\mu}^\perp\]

\[\lambda x.t \in N_{\varphi,\omega,\mu}^\perp\]

\[t \in N_{\varphi,\omega,\mu}\]

\[t \in N_{\varphi,\omega,\mu}\]

Figure 8

Local normal forms.

Notice how \(\text{id} y\) is reduced twice, which would not have happened if the second reduction had focused on the body of the abstraction.

This suggests that a substitution should only be allowed if the substituted term is in normal form. But such a strong requirement is incompatible with our calculus, as it would prevent the abstraction \(\lambda y.y\ \Omega\) (with \(\Omega\) a diverging term) to ever be substituted in the following example, thus preventing normalization (with \(a\) a closed term).

\[w (\lambda x.a)[w \backslash \lambda y.y \ \Omega] \quad \text{Iw} \quad \text{Ssn}\]

\[\lambda y.y \ \Omega\]

\[\lambda y.y \ \Omega\]

\[a[\ x \backslash \Omega][y \backslash \lambda x.a][w \backslash \lambda y.y \ \Omega]\]

Notice how the sequence of reductions has progressively removed all the occurrences of \(\Omega\), until the only term left to reduce is the closed term \(a\).

To summarize, substituting any value is too permissive and can cause duplicate computations, while substituting only normal forms is too restrictive as it prevents normalization. So, we need some relaxed notion of normal form, which we call local normal form. The intuition is as follows. The term \(\lambda y.y \ \Omega\) is not in normal form, because it could be reduced if it was in a \(\top\) position. But in a \(\bot\) position, variable \(y\) is not frozen, which prevents any further reduction of \(y \ \Omega\). The inference rules are presented in Fig. 8.

If an abstraction is in a \(\top\) position, its variable is added to the set \(\varphi\) of frozen variables, as in Fig. 3. But if an abstraction is in a \(\bot\) position, its variable is added to a new set \(\omega\), as shown in rule \(\lambda-\omega\) of Fig. 8. That is what will happen to \(y\) in \(\lambda y.y \ \Omega\).

For an application, the left part is still required to be a structure. But if the leading variable of the structure is not frozen (and thus in \(\omega\)), our \(\text{\text{\lambda}_\omega}\)-calculus guarantees that no reduction will occur in the right part of the application. So, this part does not need to be constrained in any way. This is rule \(\@-\omega\) of Fig. 8. It applies to our example, since \(y \ \Omega\) is a structure led by \(y \in \omega\). Substitutions are handled in a similar way, as shown by rule \(\text{es-}\omega\).

### 4.2 Diamond property

As mentioned before, in both \(\lambda c\) and \(\lambda_{\text{sn}}\) terms might be substituted as soon as they are values, thus potentially causing duplicate computations. As a consequence, these calculi cannot have the diamond property, as shown on the following example.
and $\lambda$

We used the Coq proof assistant for our first attempts to formalize our results. We experimented both with the locally nameless approach [13] and parametric higher-order abstract syntax [14]. While we might eventually have succeeded using the locally nameless approach,
having to manually handle binders felt way too cumbersome. So, we turned to a dedicated formal system, Abella [6], in the hope that it would make syntactic proofs more straightforward. This section describes our experience with this tool.2

5.1 Nominal variables and $\lambda$-tree syntax

Our initial motivation for using Abella was the availability of nominal variables through the $nabla$ quantifier. Indeed, in order to open a bound term, one has to replace the bound variable with a fresh global variable. This freshness is critical to avoid captures; but handling it properly causes a lot of bureaucracy in the proofs. By using nominal variables, which are guaranteed to be fresh by the logic, this issue disappears.

Here is an excerpt of our original definition of the $nf$ predicate, which states that a term is in normal form for our calculus. The second line states that any nominal variable is in normal form, while the third line states that an abstraction is in normal form, as long as the abstracted term is in normal form for any nominal variable.

```plaintext
Define $nf : trm \rightarrow prop$ by
$nabla x, nf x;$
$nabla x, nf (abs U) := nabla x, nf (U x);$ ...
```

Note that Abella is based on a $\lambda$-tree approach (higher-order abstract syntax). In the above excerpt, $U$ has a bound variable and $(U x)$ substitutes it with the fresh variable $x$. More generally, $(U V)$ is the term in which the bound variable is substituted with the term $V$.

This approach to fresh variables was error-prone at first. Several of our formalized theorems ended up being pointless, despite seemingly matching the statements of our pen-and-paper formalization. Consider the following example. This proposition states that, if $T$ is a structure with respect to $x$, and if $U$ is related to $T$ by the unfolding relation $star$, then $U$ is also a structure with respect to $x$.

```plaintext
forall $T U, nabla x,$
struct $T x \rightarrow star T U \rightarrow struct U x.$
```

Notice that the nominal variable $x$ is quantified after $T$. As a consequence, its freshness ensures that it does not occur in $T$. Thus, the proposition is vacuously true, since $T$ cannot be a structure with respect to a variable that does not occur in it. Had the quantifiers been exchanged, the statement would have been fine. Unfortunately, Abella kind of requires universal quantifiers to happen before nominal ones in theorem statements, thus exacerbating the issue. The correct way to state the above proposition is by carefully lifting any term in which a given free variable could occur:

```plaintext
forall $T U, nabla x,$
struct $(T x) x \rightarrow star (T x) (U x) \rightarrow struct (U x) x.$
```

Once one has overcome these hurdles, advantages become apparent. For example, to state that some free variable does not occur in a term, not lifting this term is sufficient. And if it needed to be lifted for some other reason, one can always equate it to a constant $\lambda$-tree. For instance, one of our theorems needs to state that the free variable $x$ occurring in $T$ cannot occur in $U$, by virtue of $star$. This is expressed as follows (with $\lambda V$ denoting $y \mapsto V$):

```plaintext
star (T x) (U x) \rightarrow exists V, U = (y \ V).
```

2 See appendix for the definitions and the statement of the main theorems, and online material for the full development.
5.2 Judgments, contexts, and derivations

Abella provides two levels of logic: a minimal logic used for specifications and an intuitionistic logic used for inductive reasoning over relations. At first, we only used the reasoning logic. By doing so, we were using Abella as if we were using Coq, except for the additional nabla quantifiers. We knew of the benefits of the specification logic when dealing with judgments and contexts; but in the case of the untyped λ-calculus, we could not see any use for those.

Our point of view started to shift once we had to manipulate sets of free variables, in order to distinguish which of them were frozen. We could have easily formalized such sets by hand; but since Abella is especially designed to handle sets of binders, we gave it a try. Let us consider the above predicate nf anew, except that it is now defined using λ-Prolog rules (π is the universal quantifier in the specification logic).

\[
\begin{align*}
\text{nff} & \colon \text{frozen} \ X.
\text{nff} (\text{abs} \ U) & \colon \pi \ x \ \text{frozen} \ x \Rightarrow \text{nff} (U x).
\end{align*}
\]

Specification-level propositions have the form \((L \ |- \ P)\), with \(P\) a proposition defined in λ-Prolog and \(L\) a list of propositions representing the context of \(P\). Consider the proposition \((L \ |- \ \text{nff} (\text{abs} \ T))\). If there were only the two rules above, there would be only three ways of deriving the proposition. Indeed, it can be derived from \((L \ |- \ \text{frozen} (\text{abs} \ T))\) (first rule). It can also be derived from nabla \(x\), \((L, \ \text{frozen} \ x \ |- \ \text{nff} (T x))\) (second rule). Finally, the third way to derive it is if \(\text{nff} (\text{abs} \ T)\) is already a member of the context \(L\).

The second and third derivations illustrate how Abella automates the handling of contexts. But where Abella shines is that some theorems come for free when manipulating specification-level properties, especially when it comes to substitution. Suppose that one wants to prove \((L \ |- \ P (T U))\), i.e., term \(T\) whose bound variable was replaced with \(U\) satisfies predicate \(P\) in context \(L\). The simplest way is if one can prove \(\text{nabla} \ x\), \((L \ |- \ P (T x))\). In that case, one can instantiate the nominal variable \(x\) with \(U\) and conclude.

But more often that not, \(x\) occurs in the context, e.g., \((L, \ Q x \ |- \ P (T x))\) instead of \((L \ |- \ P (T x))\). Then, proving \((L \ |- \ P (T U))\) is just a matter of proving \((L \ |- \ Q x)\). But, what if the latter does not hold? Suppose one can only prove \((L \ |- \ R x)\), with \(R V \ :- \ Q V\). In that case, one can reason on the derivation of \((L, \ Q x \ |- \ P (T x))\) and prove that \((L, \ R x \ |- \ P (T x))\) necessarily holds, by definition of \(R\). This ability to inductively reason on derivations is a major strength of Abella.

Having to manipulate contexts led us to revisit most of our pen-and-paper concepts. For example, a structure was no longer defined as a relation with respect to its leading variable \((\text{e.g., struct} \ T x)\) but with respect to all the frozen variables \((\text{e.g.,} \ (\text{frozen} \ x \ |- \ \text{struct} \ T))\). In turn, this led us to handle live variables purely through their addition to contexts: \(\varphi \cup \{x\}\).

Our freshness convention is a direct consequence, as in Fig. 2 for example.

Performing specification-level proofs does not come without its own set of issues, though. As explained earlier, a proposition \((L \ |- \ \text{nff} (\text{abs} \ T))\) is derivable from the consequent being part of the context \(L\), which is fruitless. The way around it is to define a predicate describing contexts that are well-formed, \(\text{e.g.,} \ L \text{ contains only propositions of the form (nf x) with x nominal. As a consequence, the case above can be eliminated because (abs T) is not a nominal variable. Unfortunately, defining these predicates and proving the associated helper lemmas is tedious and extremely repetitive. Thus, the user is encouraged to reuse existing context predicates rather than creating dedicated new ones, hence leading to sloppy and convoluted proofs. Having Abella provide some automation for handling well-formed contexts would be a welcome improvement.}
5.3 Functions and relations

Our Abella formalization assumes a type \( \text{trm} \) and three predefined ways to build elements of that type: application, abstraction, and explicit substitution. For example, a term \( t[x\backslash u] \) of our calculus will be denoted \((\text{es} (x\backslash t) u)\) with \( t \) containing some occurrences of \( x \).

\[
\begin{align*}
\text{type} & \quad \text{app} \quad \text{trm} \rightarrow \text{trm} \rightarrow \text{trm}. \\
\text{type} & \quad \text{abs} \quad (\text{trm} \rightarrow \text{trm}) \rightarrow \text{trm}. \\
\text{type} & \quad \text{es} \quad (\text{trm} \rightarrow \text{trm}) \rightarrow \text{trm} \rightarrow \text{trm}. \\
\end{align*}
\]

Since Abella does not provide functions, we instead use a relation to define the unfolding function \( t \mapsto t^\star \). Of particular interest is the way binders are handled; they are characterized by stating that they are their own image: \( \text{star} \quad x \quad x \).

\[
\begin{align*}
\text{star} (\text{app} \ U \ V) (\text{app} \ X \ Y) & :- \text{star} \ U \ X, \text{star} \ V \ Y. \\
\text{star} (\text{abs} \ U) (\text{abs} \ X) & :- \pi \ x \ \text{star} \ x \ x \Rightarrow \text{star} (U \ x) (X \ x). \\
\text{star} (\text{es} \ U \ V) (X \ Y) & :- \text{star} \ V \ Y, \pi \ x \ \text{star} \ x \ x \Rightarrow \text{star} (U \ x) (X \ x). \\
\end{align*}
\]

Since this is just a relation, we have to prove that it is defined over all the closed terms of our calculus, that it maps only to pure \( \lambda \)-terms, and that it maps to exactly one \( \lambda \)-term.

Needless to say, all of that would be simpler if Abella had native support for functions.

6 Conclusion

This paper presents a \( \lambda \)-calculus dedicated to strong reduction. In the spirit of a call-by-need strategy with explicit substitutions, it builds on a linear substitution calculus [2]. Our calculus, however, embeds a syntactic criterion that ensures that only needed redexes are considered. Moreover, by delaying substitutions until they are in so-called local normal forms rather than just values, all the reduction sequences are of minimal length.

Properly characterizing these local normal forms proved difficult and lots of iterations were needed until we reached the presented definition. Our original approach relied on evaluation contexts, as in the original presentation of a strong call-by-need strategy [9]. While tractable, this made the proof of the diamond property long and tedious. It is the use of Abella that led us to reconsider this approach. Indeed, the kind of reasoning Abella favors forced us to give up on evaluation contexts and look for reduction rules that were much more local in nature. In turn, these changes made the relation with typing more apparent. In hindsight, this would have avoided a large syntactic proof in [9].

Due to decidability, our syntactic criterion can characterize only part of the needed redexes at a given time. All the needed reductions will eventually happen, but detecting the neededness of a redex too late might prevent the optimal reduction. It is an open question whether some other simple criterion would characterize more needed redexes, and thus potentially allow for even shorter sequences than our calculus.

Even with the current criterion, there is still work to be done. First and foremost, the Abella formalization should be completed to at least include the diamond property. There are also some potential improvements to consider. For example, our calculus could be made to not substitute variables that are not applied (rule \text{lsv-base}), following [29, 3] but it opens the question of how to characterize the normal forms then. Another venue for investigation is how this work interacts with fully lazy sharing, which avoids more duplications but whose properties are tightly related to weak reduction [7]. Finally, this paper stops at describing the reduction rules of our calculus and does not investigate what an efficient abstract machine would look like.
References


A strong call-by-need calculus


A  Formal definitions

This appendix describes the main definitions of the Abella formalization. The reduction rules of $\lambda_{sn}$ and $\lambda_{sn^+}$ presented in Fig. 3 are as follows.

```prolog
step R top (abs T) (abs T') :- pi x\ frozen x => step R top (T x) (T' x).
step R B (abs T) (abs T') :- pi x\ omega x => step R bot (T x) (T' x).
step R B (app T U) (app T' U) :- step R bot B T T'.
step R B (app T U) (app T U') :- struct T, step R top U U'.
step R B (es T U) (es T' U) :- pi x\ omega x => step R B (T x) (T' x).
step R B (es T U) (es T' U) :- pi x\ frozen x => step R B (T x) (T' x), struct U.
step R B (es T U) (es T U') :- pi x\ active x => step (idx x) B (T x) (T x), step R bot U U'.
step (idx X) B X X :- active X.
step (sub X V) B X V :- active X.
step db B T T' :- aux_db T T'.
step lsv B T T' :- aux_lsv B T T'.
```

A small difference with the core of the paper is the predicate active, which characterizes the variable being considered id_x (idx) and sub_x (sub). This predicate is just a cheap way of remembering that the active variable is fresh yet not frozen. Similarly, the predicate omega is used in two rules to tag a variable as being neither frozen nor active. Another difference is rule $\lambda$-bot. While the antecedent of the rule is at position $\bot$ as in the paper, the consequent is in any position rather than just $\bot$. Since any term reducible in position $\bot$ is provably reducible in position $\top$, this is just a conservative generalization of the rule.

The auxiliary rules for $\lambda_{sn^+}$, as given in Fig. 4 and Fig. 7 for rule LSV-BASE, are the same as in the core of the paper.

```prolog
aux_db (app (abs T) U) (es T U).
aux_db (app (es T W) U) (es T' W) :- pi x\ aux_db (app (T x) U) (T' x).
aux_lsv B (es T (abs V)) (es T' (abs V)) :-
  pi x\ active x => step (sub x (abs V)) B (T x) (T' x), lnf bot (abs V).
aux_lsv B (es T (es U W)) (es T' W) :-
  pi x\ omega x => aux_lsv B (es T (U x)) (T' x).
aux_lsv B (es T (es U W)) (es T' W) :-
  pi x\ frozen x => aux_lsv B (es T (U x)) (T' x), struct W.
```

Finally, an actual reduction is just comprised of rules DB and LSV in a $\top$ position:

```prolog
red T T' :- step db top T T'.
red T T' :- step lsv top T T'.
```

The normal forms of $\lambda_{sn}$ and $\lambda_{sn^+}$, given in Fig. 5, are as follows.

```prolog
nf X :- frozen X.
nf (app U V) :- nf U, nf V, struct U.
nf (abs U) :- pi x\ frozen x => nf (U x).
nf (es U V) :- pi x\ frozen x => nf (U x), nf V, struct V.
nf (es U V) :- pi x\ nf (U x).
```

They make use of structures (struct), as given in Fig. 2.

```prolog
struct X :- frozen X.
```
A strong call-by-need calculus

```
struct (app U V) :- struct U.
struct (es U V) :- pi x\ struct (U x).
struct (es U V) :- pi x\ frozen x => struct (U x), struct V.
```

The local norm forms of Fig. 8 are as follows. As for the step relation, one of the rules for abstraction was generalized with respect to the paper. This time, it is for the \(\top\) position, since any term that is locally normal in a \(\top\) position is locally normal in any position.

```
lnf B X :- frozen X.
lnf B X :- omega X.
lnf B (app T U) :- lnf B T, struct T, lnf top U.
lnf B (app T U) :- lnf B T, struct_omega T.
lnf B (abs T) :- pi x\ frozen x => lnf top (T x).
lnf bot (abs T) :- pi x\ omega x => lnf bot (T x).
lnf B (es T U) :- pi x\ active x => lnf B (T x).
lnf B (es T U) :- pi x\ frozen x => lnf B (T x), lnf bot U, struct U.
lnf B (es T U) :- pi x\ omega x => lnf B (T x), struct_omega U.
```

Structures with respect to the set \(\omega\) use a dedicated predicate \texttt{struct_omega}, which is just a duplicate of \texttt{struct}. Another approach, perhaps more elegant, would have been to parameterize \texttt{struct} with either \texttt{frozen} or \texttt{omega}.

```
struct_omega X :- omega X.
struct_omega (app U V) :- struct_omega U.
struct_omega (es U V) :- pi x\ struct_omega (U x).
struct_omega (es U V) :- pi x\ omega x => struct_omega (U x), struct_omega V.
```

Normal forms of the \(\lambda\)-calculus are defined as follows:

```
nfb X :- frozen X.
nfb (abs T) :- pi x\ frozen x => nfb (T x).
nfb (app T U) :- nfb T, nfb U, notabs T.
notabs T :- frozen T.
```

The definition of \(\lambda_m\)-terms is sometimes useful to allow induction on terms rather than induction on one of the previous predicates.

```
trm (app U V) :- trm U, trm V.
trm (abs U) :- pi x\ trm x => trm (U x).
trm (es U V) :- pi x\ trm x => trm (U x), trm V.
```

Finally, let us remind the definitions of a pure \(\lambda\)-term, of the unfolding operation from \(\lambda\) to \(\lambda\), of a \(\beta\)-reduction, and of a sequence of zero or more \(\beta\)-reductions.

```
pure (app U V) :- pure U, pure V.
pure (abs U) :- pi x\ pure x => pure (U x).

star (app U V) (app X Y) :- star U X, star V Y.
star (abs U) (abs X) :- pi x\ star x x => star (U x) (X x).
star (es U V) (X Y) :- star V Y, pi x\ star x x => star (U x) (X x).

beta (app M N) (app M’ N) :- beta M M’.
beta (app M N) (app M N’) :- beta N N’.
beta (abs R) (abs R’) :- pi x\ beta (R x) (R’ x).
beta (app (abs R) M) (R M).
```
Formally verified properties

This appendix states the theorems that were fully proved using Abella. First comes the simulation property (Lem. 3), which states that, if $T \rightarrow_{\text{sn}^+} U$, then $T^* \rightarrow_{\beta^*} U^*$.

**Theorem** simulation' : forall $T U T^* U^*$,
\[ \{ \text{star } T T^* \} \rightarrow \{ \text{star } U U^* \} \rightarrow \{ \text{red } T U \} \rightarrow \{ \text{betas } T^* U^* \} . \]

Then comes the fact that (local) normal forms are exactly the terms that are not reducible in $\lambda_{\text{sn}^+}$ (Lem. 4).

**Theorem** lnf_nand_red : forall $T U$,
\[ \{ \text{lnf top } T \} \rightarrow \{ \text{red } T U \} \rightarrow \text{false} . \]

**Theorem** nf_nand_red : forall $T U$,
\[ \{ \text{nf } T \} \rightarrow \{ \text{red } T U \} \rightarrow \text{false} . \]

**Theorem** lnf_or_red : forall $T$,
\[ \{ \text{trm } T \} \rightarrow \{ \text{lnf top } T \} \lor \exists U, \{ \text{red } T U \} . \]

**Theorem** nf_or_red : forall $T$,
\[ \{ \text{trm } T \} \rightarrow \{ \text{nf } T \} \lor \exists U, \{ \text{red } T U \} . \]

Finally, if $T$ is a normal form of $\lambda_{\text{sn}}$, then $T^*$ is a normal form of the $\lambda$-calculus (Lem. 5).

**Theorem** nf_star' : forall $T T^*$,
\[ \{ \text{nf } T \} \rightarrow \{ \text{star } T T^* \} \rightarrow \{ \text{nfb } T^* \} . \]

Proof of the subformula properties

We recall here Lemma 8:
If $\Gamma \vdash_{\varphi} t : \tau$ and $t \in S_\varphi$, then there is $x \in \varphi$ such that $\tau \in T_\varphi(\Gamma(x))$.

**Proof.** By induction on the structure of $t$.

- Case $t = x$. By inversion of $x \in S_\varphi$ we deduce $x \in \varphi$. Moreover the only rule applicable to derive $\Gamma \vdash_{\varphi} x : \tau$ is $\text{ty-var}$, which gives the conclusion.

- Case $t = t_1 t_2$. By inversion of $t_1 t_2, t_1, t_2 \in S_\varphi$ we deduce $t_1 \in S_\varphi$. Moreover the only rules applicable to derive $\Gamma \vdash_{\varphi} t_1 t_2 : \tau$ are $\text{ty-@}$ and $\text{ty-@-S}$. Both have a premise $\Gamma' \vdash_{\varphi} t_1 : M \rightarrow \tau$ with $\Gamma' \subseteq \Gamma$, to which the induction hypothesis applies, ensuring $M \rightarrow \tau \in T_\varphi(\Gamma'(x))$ and thus $\tau \in T_\varphi(\Gamma'(x))$ and $\tau \in T_\varphi(\Gamma(x))$.

- Case $t = t_1[x \backslash t_2]$. We reason by case on the last rules applied to derive $t_1[x \backslash t_2] \in S_\varphi$ and $\Gamma \vdash_{\varphi} t_1[x \backslash t_2] : \tau$. There are two possible rules for each.
  - Case where $t_1[x \backslash t_2] \in S_\varphi$ is deduced from $t_1 \in S_\varphi$ (with $x \notin \varphi$) and $\Gamma \vdash_{\varphi} t_1[x \backslash t_2] : \tau$ comes from rule $\text{ty-es}$. This rule has in particular a premise $\Gamma' \vdash_{\varphi} t_1 : \tau$ for a $\Gamma' = \Gamma''; x : M$ such that $\Gamma'' \subseteq \Gamma$. We thus have by induction hypothesis on $t_1$ that $\tau \in T_\varphi(\Gamma''(y))$ for some $y \in \varphi \cap \text{dom}(\Gamma'')$. Since $y \in \varphi$ and $x \notin \varphi$ we have $y \neq x$. Then $y \in \text{dom}(\Gamma''')$ and $y \in \text{dom}(\Gamma)$, and $\Gamma(y) = \Gamma'''(y)$.
  - In the three other cases, we have:
    1. a hypothesis $t_1 \in S_\varphi$ or $t_1 \in S_{\varphi \cup \{x\}}$, from which we deduce $t_1 \in S_{\varphi \cup \{x\}}$,
9:22 A strong call-by-need calculus

We recall here Lemma 9:

1. If \( \Phi \vdash t : \tau \) then \( \mathcal{T}_+ (fzt(\Phi)) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(x)) \cup \mathcal{T}_-(\tau) \)
2. If \( \Phi \vdash \Gamma \vdash t : \tau \) then \( \mathcal{T}_- (fzt(\Phi)) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_- (\Gamma(x)) \cup \mathcal{T}_+ (\tau) \)

Proof. By mutual induction on the typing derivations.

Case for abstraction.

If \( \Phi \vdash \Gamma \vdash \lambda x.t : \mathcal{M} \rightarrow \tau \) by rule TY-L-\perp with premise \( \Phi' \vdash \Gamma; x : \mathcal{M} \vdash t : \tau \).
Write \( \Gamma' = \Gamma; x : \mathcal{M} \). By induction hypothesis we have \( \mathcal{T}_+ (fzt(\Phi')) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(y)) \).
Since \( \varphi \not\subseteq \tau \) by renaming convention, we deduce that \( \mathcal{T}_+ (fzt(\Phi')) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(y)) \).
This implies \( \mathcal{T}_+ (fzt(\Phi)) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(y)) \).
The same applies to negative type occurrences, which concludes the case.

Case for application.

If \( \Phi \vdash \Gamma \vdash t : \mathcal{M} \rightarrow \tau \) by rule TY-@ with premise \( \Phi' \vdash \Gamma; \mathcal{M} \vdash \tau \).
Write \( \Gamma' = \Gamma; \mathcal{M} \). By induction hypothesis we have \( \mathcal{T}_+ (fzt(\Phi')) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(y)) \).
Independently of the value of \( \mu \), we show that \( \mathcal{T}_+ (fzt(\Phi)) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(x)) \) and \( \mathcal{T}_- (fzt(\Phi')) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_- (\Gamma(x)) \). For each \( \phi \in \varphi \), we have \( \mathcal{T}_+ (fzt(\Phi)) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(x)) \).
Thus \( \mathcal{T}_+ (fzt(\Phi)) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(x)) \) and \( \mathcal{T}_- (\mathcal{M} \rightarrow \tau) \). The same applies to negative type occurrences, which concludes the case.

Case for explicit substitution immediately follow the induction hypothesis.

We recall here Lemma 9:

1. If \( \Phi \vdash t : \tau \) then \( \mathcal{T}_+ (fzt(\Phi)) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(x)) \cup \mathcal{T}_-(\tau) \)
2. If \( \Phi \vdash \Gamma \vdash t : \tau \) then \( \mathcal{T}_- (fzt(\Phi)) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_- (\Gamma(x)) \cup \mathcal{T}_+ (\tau) \)

Proof. By mutual induction on the typing derivations.

Case for abstractions.

If \( \Phi \vdash \Gamma \vdash \lambda x.t : \mathcal{M} \rightarrow \tau \) by rule TY-L-\perp with premise \( \Phi' \vdash \Gamma; x : \mathcal{M} \vdash t : \tau \).
Write \( \Gamma' = \Gamma; x : \mathcal{M} \). By induction hypothesis we have \( \mathcal{T}_+ (fzt(\Phi')) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(y)) \).
Since \( \varphi \not\subseteq \tau \) by renaming convention, we deduce that \( \mathcal{T}_+ (fzt(\Phi')) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(y)) \).
This implies \( \mathcal{T}_+ (fzt(\Phi)) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(y)) \).
The same applies to negative type occurrences, which concludes the case.

Case for application.

If \( \Phi \vdash \Gamma \vdash t : \mathcal{M} \rightarrow \tau \) by rule TY-@ with premise \( \Phi' \vdash \Gamma; \mathcal{M} \vdash \tau \).
Write \( \Gamma' = \Gamma; \mathcal{M} \). By induction hypothesis we have \( \mathcal{T}_+ (fzt(\Phi')) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(x)) \).
Independently of the value of \( \mu \), we show that \( \mathcal{T}_+ (fzt(\Phi)) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(x)) \) and \( \mathcal{T}_- (fzt(\Phi')) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_- (\Gamma(x)) \). For each \( \phi \in \varphi \), we have \( \mathcal{T}_+ (fzt(\Phi)) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(x)) \).
Thus \( \mathcal{T}_+ (fzt(\Phi)) \subseteq \bigcup_{\varphi \in \varphi} \mathcal{T}_+ (\Gamma(x)) \) and \( \mathcal{T}_- (\mathcal{M} \rightarrow \tau) \). The same applies to negative type occurrences, which concludes the case.

Case for explicit substitution immediately follow the induction hypothesis.