A STRONG CALL-BY-NEED CALCULUS

THIBAUT BALABONSKI [a], ANTOINE LANCO [b], AND GUILLAUME MELQUIOND [b]

[a] Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, Gif-sur-Yvette, 91190, France
e-mail address: thibaut.balabonski@lri.fr

[b] Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, Gif-sur-Yvette, 91190, France
e-mail address: antoine.lanco@lri.fr
e-mail address: guillaume.melquiond@inria.fr

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. In particular, it led us to derive a new description of call-by-need reduction based on inductive rules.

1. Introduction

Lambda-calculus is seen as the standard model of computation in functional programming languages, once equipped with an *evaluation strategy* [Plo75]. 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* [Wad71, AMO+95], 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.

Key words and phrases: strong reduction, call-by-need, evaluation strategy, normalization.

Preprint submitted to
Logical Methods in Computer Science

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 [Cré90] 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 [GL02, BBCD20, ACSC21], call-by-name [ABM15], and call-by-need [BBBK17, BC19]. 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 [Bal13], it is known that the deliberate use of strong reduction [HG91] 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, *i.e.*, redexes that are necessarily reduced in any β-reduction to normal form of the underlying λ-term. 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 λ-calculus with explicit substitutions, which we recall in Section 2.

We strayed away from the more traditional style of presentation of weak call-by-need reduction, based on evaluation contexts, and went for SOS-style reduction rules [Plo81], which later became a lever for subsequent contributions of this work. Since this style of presentation is itself a contribution, we devote Section 3 to it.

Neededness of a redex is undecidable in general, thus the first and third guidelines above are antagonistic. One cannot reduce only needed redexes while enabling strong reduction as generally as possible. Consider for instance a λ-term like $(\lambda x.t)\ (\lambda y.r)$, where r is a redex. Strong reduction could allow reducing the redex r before substituting the value $\lambda y.r$, which might be particularly interesting to prevent r from being reduced multiple times after substitution. However, we certainly do not want to do that if r is a diverging term and the normal form of $(\lambda x.t)\ (\lambda y.r)$ can be computed without reducing r.

Section 4 resolves this tension by exposing a simple syntactic criterion capturing more needed redexes than what is already used in call-by-need strategies. By reducing needed redexes only, our calculus enjoys a normalization preservation theorem that is stronger than usual: Any λ-term that is *weakly* normalizing in the pure λ-calculus (*i.e.*, there is at least one reduction sequence to a normal form, but some other sequences may diverge) will be *strongly* normalizing in our calculus (*i.e.*, 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 [Kes16, BBBK17], enabled by our SOS-style presentation.

While our calculus contains the strong call-by-need strategy introduced in [BBBK17], 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 5 describes a restriction of the calculus that guarantees reduction sequences of minimal length.

Section 6 presents a formalization of parts of our results in Abella [BCG$^+$14]. 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, this is what promoted the aforementioned presentation based on SOS-style local reduction rules. The formalization can be found at the following address: `https://hal.inria.fr/hal-03149692`.

Finally, Section 7 presents an abstract machine that implements the rules of our calculus. More precisely, it implements the leftmost strategy that complies with our calculus, using a mutable store to deal with explicit substitutions in a call-by-need fashion. This section presents the big-step semantics of the machine, and draws a few observations from it.

This paper is an extended version of [BLM21]. It contains more examples, more explanations, and more detailed proofs than the short paper, as well as two new sections. First, Section 3 details the technique used for expressing call-by-need reduction in SOS-style, and gives a proof that this presentation actually defines the same calculus as the more traditional presentation of weak call-by-need by means of evaluation contexts. Second, Section 7 on the abstract machine is fully new material. Finally, we provide in this extended version a formal statement of how this strong call-by-need calculus indeed reduces only needed redexes.

2. The host calculus λ_{C}

Our strong call-by-need calculus is included in an already known calculus λ_{c}, that serves as a technical tool in [BBBK17] 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 λ_{c}-calculus uses the following syntax of λ-terms with explicit substitutions, which is isomorphic to the original syntax of the call-by-need calculus using let-bindings [AMO$^+$95].

$$t \in \Lambda_{\mathsf{c}} \quad ::= \quad x \mid \lambda x.t \mid t\ t \mid t[x\backslash t]$$

The free variables $\mathsf{fv}(t)$ of a term t are defined as usual. We call pure λ-term a term that contains no explicit substitution. We write $\mathcal{C}(\bullet)$ for a context, *i.e.*, a term with exactly one hole $\bullet$, and $\mathcal{L}$ for a context with the specific shape $\bullet[x_1\backslash t_1]\dots[x_n\backslash t_n]$ ($n \geqslant 0$). We write $\mathcal{C}(t)$ for the term obtained by plugging the subterm t in the hole of the context $\mathcal{C}(\bullet)$, with possible capture of free variables of t by binders in $\mathcal{C}(\bullet)$, or $t\mathcal{L}$ when the context is of the specific shape $\mathcal{L}$. We also write $\mathcal{C}(\!|t|\!)$ for plugging a term t whose free variables are not captured by $\mathcal{C}(\bullet)$.

The *values* we consider are λ-abstractions, and are usually denoted by v. If variables were also considered as values, most of the results of this paper would be preserved, except for the diamond property in the restricted calculus (see Theorem 5.2).

Reduction in λ_{c} is defined by the following three reduction rules, applied in any context. Rather than using traditional propagation rules for explicit substitutions [Kes09], it builds on the *Linear Substitution Calculus* [Mil07, AK10, ABKL14] which is more similar to the

$$\text{TY-VAR}\ \frac{}{x : \{\!\{\sigma\}\!\} \vdash x : \sigma} \qquad \text{TY-APP}\ \frac{\Gamma \vdash t : \mathcal{M} \to \tau \qquad (\Delta_\sigma \vdash u : \sigma)_{\sigma \in \mathcal{M}}}{\Gamma + \sum_{\sigma \in \mathcal{M}} \Delta_\sigma \vdash t\ u : \tau}$$

$$\text{TY-ABS}\ \frac{\Gamma, x : \mathcal{M} \vdash t : \tau}{\Gamma \vdash \lambda x.t : \mathcal{M} \to \tau} \qquad \text{TY-ES}\ \frac{\Gamma, x : \mathcal{M} \vdash t : \tau \qquad (\Delta_\sigma \vdash u : \sigma)_{\sigma \in \mathcal{M}}}{\Gamma + \sum_{\sigma \in \mathcal{M}} \Delta_\sigma \vdash t[x\backslash u] : \tau}$$

Figure 1: Typing rules for λ_{c}.

let-in constructs commonly used for defining call-by-need.

$$\begin{array}{rcll} (\lambda x.t)\mathcal{L}\ u & \to_{\mathsf{dB}} & t[x\backslash u]\mathcal{L} & \\ \mathcal{C}(\!|x|\!)[x\backslash v\mathcal{L}] & \to_{\mathsf{lsv}} & \mathcal{C}(\!|v|\!)[x\backslash v]\mathcal{L} & \text{with } v \text{ value,} \\ t[x\backslash u] & \to_{\mathsf{gc}} & t & \text{with } x \notin \mathsf{fv}(t). \end{array}$$

The rule $\to_{\mathsf{dB}}$ describes β-reduction "at a distance". It applies to a β-redex whose λ-abstraction is possibly hidden by a list $\mathcal{L}$ of explicit substitutions. This rule is a combination of a single use of β-reduction with a repeated use of the structural rule lifting the explicit substitutions at the left of an application. The rule $\to_{\mathsf{lsv}}$ 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 $\to_{\mathsf{dB}}$, this rule embeds a repeated use of a structural rule for unnesting explicit substitutions. Note that this calculus only allows the substitution of λ-abstractions, and not of variables as it is sometimes seen [MOW98]. This restricted behavior is enough for the main results of this paper, and will allow a more compact presentation. Finally, the rule $\to_{\mathsf{gc}}$ 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 $t \to_{\mathsf{c}} u$.

A λ_{c}-term t is related to a pure λ-term $t^\star$ by the unfolding operation which applies all the explicit substitutions. We write $t\{x\backslash u\}$ for the meta-level substitution of x by u in t.

$$\begin{array}{rcl@{\qquad}rcl} x^\star & = & x & (t\ u)^\star & = & t^\star\ u^\star \\ (\lambda x.t)^\star & = & \lambda x.(t^\star) & (t[x\backslash u])^\star & = & (t^\star)\{x\backslash u^\star\} \end{array}$$

Through unfolding, any reduction step $t \to_{\mathsf{c}} u$ in the λ_{c}-calculus is related to a sequence of reductions $t^\star \to_\beta^* u^\star$ in the pure λ-calculus.

The host calculus λ_{c} comes with a system of non-idempotent intersection types [CDC80, Gar94], defined in [KV14] by adding explicit substitutions to an original system from [Gar94]. A type τ may be a type variable α or an arrow type $\mathcal{M} \to \tau$, where $\mathcal{M}$ is a multiset $\{\!\{\sigma_1, \ldots, \sigma_n\}\!\}$ of types.

A typing environment Γ 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. In the latter case, the variable can simply be removed from the typing environment, *i.e.*, we equate Γ with $\Gamma, x : \{\!\{\}\!\}$.

A typing judgment $\Gamma \vdash t : \tau$ assigns exactly one type to the term t. As shown by the typing rules in Figure 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 $\sigma \in \mathcal{M}$ quantifies on all the instances of elements in the multiset $\mathcal{M}$.

Example. Given $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$, the term $(\lambda xy.x\ (x\ x))\ (\lambda z.z)\ \Omega$ is typable in this system, while Ω itself is not. Let σ_0 be a type. Write $\sigma_1 = \{\!\{\sigma_0\}\!\} \to \sigma_0$ and $\sigma_2 = \{\!\{\sigma_1\}\!\} \to \sigma_1$. First we derive the judgment $\vdash \lambda xy.x\ (x\ x) : \{\!\{\sigma_1, \sigma_2, \sigma_2\}\!\} \to \{\!\{\}\!\} \to \sigma_1$ as follows. The main subtlety of the derivation is the adequate splitting of the environment where the application rule is used.

$$\dfrac{\dfrac{\dfrac{}{x : \{\!\{\sigma_2\}\!\} \vdash x : \{\!\{\sigma_1\}\!\} \to \sigma_1}\ \text{TY-VAR} \quad \dfrac{\dfrac{}{x : \{\!\{\sigma_2\}\!\} \vdash x : \{\!\{\sigma_1\}\!\} \to \sigma_1}\ \text{TY-VAR} \quad \dfrac{}{x : \{\!\{\sigma_1\}\!\} \vdash x : \sigma_1}\ \text{TY-VAR}}{x : \{\!\{\sigma_1, \sigma_2\}\!\} \vdash x\ x : \sigma_1}\ \text{TY-APP}}{x : \{\!\{\sigma_1, \sigma_2, \sigma_2\}\!\}, y : \{\!\{\}\!\} \vdash x\ (x\ x) : \sigma_1}\ \text{TY-APP}}{x : \{\!\{\sigma_1, \sigma_2, \sigma_2\}\!\} \vdash \lambda y.x\ (x\ x) : \{\!\{\}\!\} \to \sigma_1}\ \text{TY-ABS}}{\vdash \lambda xy.x\ (x\ x) : \{\!\{\sigma_1, \sigma_2, \sigma_2\}\!\} \to \{\!\{\}\!\} \to \sigma_1}\ \text{TY-ABS}$$

From this first judgment, we deduce a typing derivation for $(\lambda xy.x\ (x\ x))\ (\lambda z.z)\ \Omega$. This derivation contains three type derivations for the first argument $\lambda z.z$ and no type derivation for the second argument Ω. The three type derivations for $\lambda z.z$ are for each of the types in the multiset $\{\!\{\sigma_1, \sigma_2, \sigma_2\}\!\}$. Two of them are identical.

$$\dfrac{\dfrac{\ldots \quad \dfrac{\dfrac{}{z : \{\!\{\sigma_0\}\!\} \vdash z : \sigma_0}\ \text{TY-VAR}}{\vdash \lambda z.z : \sigma_1}\ \text{TY-ABS} \quad \dfrac{\dfrac{}{z : \{\!\{\sigma_1\}\!\} \vdash z : \sigma_1}\ \text{TY-VAR}}{\vdash \lambda z.z : \sigma_2}\ \text{TY-ABS} \quad \dfrac{\dfrac{}{z : \{\!\{\sigma_1\}\!\} \vdash z : \sigma_1}\ \text{TY-VAR}}{\vdash \lambda z.z : \sigma_2}\ \text{TY-ABS}}{\vdash (\lambda xy.x(xx))\ (\lambda z.z) : \{\!\{\}\!\} \to \sigma_1}\ \text{TY-APP}}{\vdash (\lambda xy.x(xx))\ (\lambda z.z)\ \Omega : \sigma_1}\ \text{TY-APP}$$

This type system is known to characterize λ-terms that are weakly normalizing for β-reduction, if associated with the side condition that the empty multiset $\{\!\{\}\!\}$ does not appear at a positive position in the typing judgment $\Gamma \vdash t : \tau$. Positive and negative occurrences of types are defined by the following equations.

$$\begin{array}{rclcrcl}
\mathcal{T}_+(\alpha) & = & \{\alpha\} & & \mathcal{T}_-(\alpha) & = & \emptyset \\
\mathcal{T}_+(\mathcal{M}) & = & \{\mathcal{M}\} \cup \bigcup_{\sigma \in \mathcal{M}} \mathcal{T}_+(\sigma) & & \mathcal{T}_-(\mathcal{M}) & = & \bigcup_{\sigma \in \mathcal{M}} \mathcal{T}_-(\sigma) \\
\mathcal{T}_+(\mathcal{M} \to \sigma) & = & \{\mathcal{M} \to \sigma\} \cup \mathcal{T}_-(\mathcal{M}) \cup \mathcal{T}_+(\sigma) & & \mathcal{T}_-(\mathcal{M} \to \sigma) & = & \mathcal{T}_+(\mathcal{M}) \cup \mathcal{T}_-(\sigma) \\
\mathcal{T}_+(\Gamma \vdash t : \sigma) & = & \mathcal{T}_+(\sigma) \cup \bigcup_{x \in \mathsf{dom}(\Gamma)} \mathcal{T}_-(\Gamma(x)) & & & &
\end{array}$$

Theorem 2.1 (Typability [dC07, BKV17])**.** *If the pure λ-term t is weakly normalizing for β-reduction, then there is a typing judgment $\Gamma \vdash t : \tau$ such that $\{\!\{\}\!\} \notin \mathcal{T}_+(\Gamma \vdash t : \tau)$.*

Example. Let α, β, γ, and δ be type variables. Positive and negative type occurrences of the type $\tau = \{\!\{\{\!\{\alpha, \beta\}\!\} \to \gamma\}\!\} \to \{\!\{\}\!\} \to \delta$ are as follows:

$$\begin{array}{rcl}
\mathcal{T}_+(\tau) & = & \{\{\!\{\{\!\{\alpha, \beta\}\!\} \to \gamma\}\!\} \to \{\!\{\}\!\} \to \delta, \{\!\{\alpha, \beta\}\!\}, \alpha, \beta, \{\!\{\}\!\} \to \delta, \delta\} \\
\mathcal{T}_-(\tau) & = & \{\{\!\{\{\!\{\alpha, \beta\}\!\} \to \gamma\}\!\}, \{\!\{\alpha, \beta\}\!\} \to \gamma, \gamma, \{\!\{\}\!\}\}
\end{array}$$

A typing derivation Φ for a typing judgment $\Gamma \vdash t : \tau$ (written $\Phi \rhd \Gamma \vdash t : \tau$) defines in t a set of *typed positions*, which are the positions of the subterms of t for which the derivation Φ contains a subderivation. More precisely:

- ε is a typed position for any derivation;
- if Φ ends with rule TY-ABS, TY-APP or TY-ES, then $0p$ is a typed position of Φ if p is a typed position of the subderivation Φ' relative to the first premise;

- if Φ ends with rule TY-APP or TY-ES, then $1p$ is a typed position of Φ if p is a typed position of the subderivation Φ' 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 $\mathcal{M}$ is empty. On the contrary, when $\mathcal{M}$ has several elements, we get the union of the typed positions contributed by each instance.

Example. In the derivation proposed above for the judgment $\vdash (\lambda xy.x(xx))\ (\lambda z.z)\ \Omega : \sigma_1$, the position ε, and all the positions of the term starting with 0 are typed. On the contrary, no position starting with 1 is typed.

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 Φ the number of nodes of the derivation tree.

Theorem 2.2 (Weighted subject reduction [BBBK17])**.** *If $\Phi \rhd \Gamma \vdash t : \tau$ and $t \to_{\mathsf{c}} t'$ by reduction of a redex at a typed position, then there is a derivation $\Phi' \rhd \Gamma \vdash t' : \tau$ with Φ' smaller than Φ.*

Note also that typing is preserved by reverting explicit substitutions to β-redexes.

Proposition 2.3 (Typing and substitution)**.** *$\Gamma \vdash (\lambda x.t)\ u : \tau$ if and only if $\Gamma \vdash t[x\backslash u] : \tau$.*

Thus, if we write $t^\dagger$ the pure λ-term obtained by reverting all the explicit substitutions of a λ_{c}-term t, which can be defined by the following equations,

$$\begin{array}{rclcrcl} x^\dagger & = & x & \qquad & (t\ u)^\dagger & = & t^\dagger\ u^\dagger \\ (\lambda x.t)^\dagger & = & \lambda x.(t^\dagger) & & (t[x\backslash u])^\dagger & = & (\lambda x.t^\dagger)\ u^\dagger \end{array}$$

then we have the following theorem.

Proposition 2.4 (Stability of typing by reverting explicit substitutions)**.** *For any term t, $\Gamma \vdash t : \tau$ if and only if $\Gamma \vdash t^\dagger : \tau$.*

Proof. By induction on t. □

Given any λ_{c}-term t, the reverted term $t^\dagger$ can be related to the unfolded term $t^\star$ by β-reducing precisely the β-redexes introduced by the † operation. This is proved by a straightforward induction on t.

Proposition 2.5. *For any term t, $t^\dagger \to_\beta^* t^\star$.*

Hence $t^\dagger$ is weakly normalizing, and thus t is typable, if and only if $t^\star$ is weakly normalizing.

3. Weak call-by-need, revisited

Call-by-need is usually defined using a formalism that is equivalent or equal to λ_{c}, by restricting the situations in which the reduction rules can be applied. The main idea is that since the argument of a function is not always needed, we do not reduce in advance the right part of an application $t\ u$. Instead, we first evaluate t to an answer $(\lambda x.t')\mathcal{L}$ (a value under some explicit substitutions), then apply a dB-reduction to put the argument u in the environment of t', and then go on with the resulting term $t'[x\backslash u]\mathcal{L}$, evaluating u only if and when it is required.

Call-by-need using evaluation contexts. This restriction on the reduced redexes is traditionally defined in weak call-by-need through a notion of evaluation context [AMO+95, CF12]. Such a definition using the syntax of our host calculus λ_{c} can for instance be found in [ABM14]. Evaluation contexts $\mathcal{E}(\bullet)$ are characterized by the following grammar:

$$\mathcal{E}(\bullet) \quad ::= \quad \bullet \mid \mathcal{E}(\bullet)\ u \mid \mathcal{E}(\bullet)[x\backslash u] \mid \mathcal{E}(\!|x|\!)[x\backslash\mathcal{E}(\bullet)]$$

In this presentation, the dB rule is used exactly as it is presented in λ_{c}, and the lsv rule is restricted to allow substitution only for an occurrence of x that is reachable through an evaluation context. The rule gc is not used, since it is never actually necessary.

$$\begin{array}{rcll} (\lambda x.t)\mathcal{L}\ u & \mapsto_{\mathsf{dB}} & t[x\backslash u]\mathcal{L} & \\ \mathcal{E}(\!|x|\!)[x\backslash v\mathcal{L}] & \mapsto_{\mathsf{lsv}} & \mathcal{E}(\!|v|\!)[x\backslash v]\mathcal{L} & \text{with } v \text{ value.} \end{array}$$

Then t is defined to reduce to t' for a base rule ρ whenever there exists an evaluation context $\mathcal{E}(\bullet)$ and a ρ-redex r such that $t = \mathcal{E}(r)$, $r \mapsto_\rho r'$ and $t' = \mathcal{E}(r')$. Let us call λ_{wn} the weak call-by-need calculus defined by these rules and contexts.

Example. If v is a value, then $(y\ u)[y\backslash v[z\backslash w]] \mapsto_{\mathsf{lsv}} (v\ u)[y\backslash v][z\backslash w]$ is an instance of the base reduction rule lsv. Indeed, $(y\ u)[y\backslash v[z\backslash w]]$ can be written $\mathcal{E}(\!|y|\!)[y\backslash v\mathcal{L}]$ with $\mathcal{E}(\bullet) = \bullet\ u$ an evaluation context and $\mathcal{L} = [z\backslash w]$ a list of explicit substitutions. Moreover $\mathcal{E}'(\bullet) = (x\ t)[x\backslash\bullet]$ is an evaluation context, built with the shape $\mathcal{E}'_1(\!|x|\!)[x\backslash\mathcal{E}'_2(\bullet)]$ with $\mathcal{E}'_1(\bullet) = \bullet\ t$ and $\mathcal{E}'_2(\bullet) = \bullet$. Thus by applying the previous $\mapsto_{\mathsf{lsv}}$-reduction in the evaluation context $\mathcal{E}'(\bullet)$, we justify the following reduction step:

$$(x\ t)[x\backslash(y\ u)[y\backslash v[z\backslash w]]] \to (x\ t)[x\backslash(v\ u)[y\backslash v][z\backslash w]]$$

This presentation based on evaluation contexts is difficult to handle. A notable issue is the formalization of the plugging operations $\mathcal{E}(t)$ and $\mathcal{E}(\!|t|\!)$ and their relation with α-equivalence. Hence we propose here a new presentation of weak call-by-need reduction based on an equivalent inductive definition, which we will extend in the rest of the paper to define and formalize strong call-by-need reduction.

Inductive definition of call-by-need reduction. We provide an alternative definition of λ_{wn}, based on an inductive definition of a binary relation $\xrightarrow{\rho}$ on terms, parameterized by a rule ρ which can be dB, lsv, or two others that we will introduce shortly. The inference rules are given in Figure 2.

Rule APP-LEFT makes reduction always possible on the left of an application, and rule ES-LEFT does the same for terms that are under an explicit substitution. They correspond respectively to the shapes $\mathcal{E}(\bullet)\ u$ and $\mathcal{E}(\bullet)[x\backslash u]$ of evaluation contexts. Remark that there is no inference rule allowing reduction inside a λ-abstraction. Therefore, this truly defines only weak reduction steps.

Rule ES-RIGHT restricts reduction of the argument of a substitution to the case where an occurrence of the variable affected by the substitution is at a reducible position, as did the evaluation context shape $\mathcal{E}(\!|x|\!)[x\backslash\mathcal{E}(\bullet)]$. In order to probe a term for the presence of some variable x at a reducible position, which is what the condition $\mathcal{E}(\!|x|\!)$ means, without mentioning any evaluation context, it uses an auxiliary rule id_x which propagates using the same inductive reduction relation. This auxiliary reduction rule does not modify the term to which it applies, as witnessed by its base case ID.

$$\textsc{app-left}\ \frac{t \xrightarrow{\rho} t'}{t\ u \xrightarrow{\rho} t'\ u} \qquad \textsc{es-left}\ \frac{t \xrightarrow{\rho} t'}{t[x\backslash u] \xrightarrow{\rho} t'[x\backslash u]} \qquad \textsc{es-right}\ \frac{t \xrightarrow{\mathsf{id}_x} t \qquad u \xrightarrow{\rho} u'}{t[x\backslash u] \xrightarrow{\rho} t[x\backslash u']}$$

$$\textsc{id}\ \frac{}{x \xrightarrow{\mathsf{id}_x} x} \qquad \textsc{sub}\ \frac{}{x \xrightarrow{\mathsf{sub}_{x\backslash v}} v} \qquad \textsc{dB}\ \frac{t \to_{\mathsf{dB}} t'}{t \xrightarrow{\mathsf{dB}} t'} \qquad \textsc{lsv}\ \frac{t \to_{\mathsf{lsv}} t'}{t \xrightarrow{\mathsf{lsv}} t'}$$

Figure 2: Reduction rules for λ_{wn}.

$$\textsc{dB-base}\ \frac{}{(\lambda x.t)\ u \to_{\mathsf{dB}} t[x\backslash u]} \qquad \textsc{lsv-base}\ \frac{t \xrightarrow{\mathsf{sub}_{x\backslash v}} t' \qquad v \text{ value}}{t[x\backslash v] \to_{\mathsf{lsv}} t'[x\backslash v]}$$

$$\textsc{dB-}\sigma\ \frac{t\ u \to_{\mathsf{dB}} s}{t[x\backslash w]\ u \to_{\mathsf{dB}} s[x\backslash w]} \qquad \textsc{lsv-}\sigma\ \frac{t[x\backslash u] \to_{\mathsf{lsv}} t'}{t[x\backslash u[y\backslash w]] \to_{\mathsf{lsv}} t'[y\backslash w]}$$

Figure 3: Auxiliary reduction rules for λ_{wn}.

Rules dB and lsv are the base cases for applying reductions dB or lsv. Using the context notations, they allow the following reductions.

$$\begin{array}{rcll} (\lambda x.t)\mathcal{L}\ u & \xrightarrow{\mathsf{dB}} & t[x\backslash u]\mathcal{L} & \\ \mathcal{E}(\!|x|\!)[x\backslash v\mathcal{L}] & \xrightarrow{\mathsf{lsv}} & \mathcal{E}(\!|v|\!)[x\backslash v]\mathcal{L} & \text{with } v \text{ value.} \end{array}$$

However, we are aiming for a definition that does not rely on the notion of context. We have to express lsv-reduction without the evaluation context $\mathcal{E}(\bullet)$. Moreover, a list $\mathcal{L}$ of explicit substitutions *is* a context and thus brings the same formalization problems as evaluation contexts. Therefore we define each of these rules using an auxiliary (inductive) reduction relation dealing with the list $\mathcal{L}$ of explicit substitutions and, in the case of lsv, with the evaluation context $\mathcal{E}(\bullet)$. The inference rules for these auxiliary relations are given in Figure 3.

Rules dB-base and lsv-base describe the base cases of the auxiliary reductions, where the list $\mathcal{L}$ is empty. Note that, while dB-base is an axiom, the inference rule lsv-base uses as a premise a reduction $\xrightarrow{\rho}$ using a new reduction rule $\rho = \mathsf{sub}_{x\backslash v}$. This rule substitutes one occurrence of the variable x at a reducible position by the value v, and implements the transformation of $\mathcal{E}(\!|x|\!)$ into $\mathcal{E}(\!|v|\!)$. As seen for id_x above, this reduction rule propagates using the same inductive reduction relation, and its base case is the rule sub in Figure 2.

Rule dB-σ makes it possible to float out an explicit substitution applied to the left part of an application. That is, if a dB-reduction is possible without the substitution, then the reduction is performed and the substitution is applied to the result. Rule lsv-σ achieves the same effect with the nested substitutions applied to the value substituted by an lsv-reduction step. In other words, each of these two rules applied repeatedly moves an arbitrary list $\mathcal{L}$ of substitutions until the corresponding base case becomes applicable.

Finally, we say t reduces to t' whenever we can derive $t \xrightarrow{\rho} t'$ with $\rho \in \{\mathsf{dB}, \mathsf{lsv}\}$.

Example. The reduction $(x\ t)[x\backslash(y\ u)[y\backslash v[z\backslash w]]] \xrightarrow{\mathsf{lsv}} (x\ t)[x\backslash(v\ u)[y\backslash v][z\backslash w]]$ presented in the previous example is also derivable with these reduction rules. The derivation contains two main branches. The left branch checks that an occurrence of x is reachable in the main term $x\ t$, using the relation $\xrightarrow{\mathsf{id}_x}$. The right branch then performs reduction in the argument of the substitution. This right branch immediately invokes the auxiliary reduction $\to_{\mathsf{lsv}}$, to conclude in two steps: first moving out the substitution $[z\backslash w]$ which hides the value v, using a σ-rule, and then applying a linear substitution of y by v in the term $y\ u$, using the relation $\xrightarrow{\mathsf{sub}_{y\setminus v}}$.

$$\dfrac{\dfrac{\text{ID}\ \dfrac{}{x \xrightarrow{\mathsf{id}_x} x}}{\text{APP-LEFT}\ \dfrac{x\ t \xrightarrow{\mathsf{id}_x} x\ t}{}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\text{SUB}\ \dfrac{}{y \xrightarrow{\mathsf{sub}_{y\setminus v}} v}}{\text{APP-LEFT}\ \dfrac{y\ u \xrightarrow{\mathsf{sub}_{y\setminus v}} v\ u}{}} \qquad v\ \text{value}}{(y\ u)[y\backslash v] \to_{\mathsf{lsv}} (v\ u)[y\backslash v]}\ \text{LSV-BASE}}{(y\ u)[y\backslash v[z\backslash w]] \to_{\mathsf{lsv}} (v\ u)[y\backslash v][z\backslash w]}\ \text{LSV-}\sigma}{(y\ u)[y\backslash v[z\backslash w]] \xrightarrow{\mathsf{lsv}} (v\ u)[y\backslash v][z\backslash w]}\ \text{LSV}}{(x\ t)[x\backslash(y\ u)[y\backslash v[z\backslash w]]] \xrightarrow{\mathsf{lsv}} (x\ t)[x\backslash(v\ u)[y\backslash v][z\backslash w]]}\ \text{ES-RIGHT}$$

Theorem 3.1 (Equivalence)**.** *We have the following equivalences between the inductive definition of λ_{wn} and the evaluation context-based presentation.*

(1) $t \xrightarrow{dB} t'$ *iff there are* $\mathcal{E}(\bullet)$, r, *and* r' *such that* $t = \mathcal{E}(r)$, $r \mapsto_{dB} r'$, *and* $t' = \mathcal{E}(r')$;
(2) $t \xrightarrow{lsv} t'$ *iff there are* $\mathcal{E}(\bullet)$, r, *and* r' *such that* $t = \mathcal{E}(r)$, $r \mapsto_{lsv} r'$, *and* $t' = \mathcal{E}(r')$;
(3) $t \xrightarrow{id_x} t$ *iff there is* $\mathcal{E}(\bullet)$ *such that* $t = \mathcal{E}(\!|x|\!)$;
(4) $t \xrightarrow{sub_{x\setminus v}} t'$ *iff there is* $\mathcal{E}(\bullet)$ *such that* $t = \mathcal{E}(\!|x|\!)$ *and* $t' = \mathcal{E}(\!|v|\!)$.

The proof of this theorem also requires statements relating the auxiliary reduction relations $\to_{\mathsf{dB}}$ and $\to_{\mathsf{lsv}}$ to the base reduction rules $\mapsto_{\mathsf{dB}}$ and $\mapsto_{\mathsf{lsv}}$.

Lemma 3.2 (Equivalence)**.** *We have the following equivalences between the inductive definition of the auxiliary rules of λ_{wn} and the base reduction rules of the evaluation context-based presentation.*

(1) $t \to_{dB} t'$ *iff* $t \mapsto_{dB} t'$;
(2) $t \to_{lsv} t'$ *iff* $t \mapsto_{lsv} t'$.

The second equivalence has to be proved in a mutual induction with the main theorem, since the inference rules for $\to_{\mathsf{lsv}}$ use $\xrightarrow{\rho}$ and the base rule $\mapsto_{\mathsf{lsv}}$ uses an evaluation context. The first equivalence however is a standalone property.

Proof of Lemma 3.2 item 1. The implication from left to right is a simple induction on the derivation of $t \to_{\mathsf{dB}} t'$, while the reverse implication is proved by induction on the length of the list $\mathcal{L}$. □

Proof of Theorem 3.1 and Lemma 3.2 item 2. The five equivalences are proved together. Implications from left to right are proved by a single induction on the derivation of $t \xrightarrow{\rho} t'$ or $t \to_{\mathsf{lsv}} t'$, whereas implications from right to left are proved by a single induction on the evaluation context $\mathcal{E}(\bullet)$ and on the length of the list $\mathcal{L}$. In the detailed proof, we extend the notation $\mapsto_\rho$ for any ρ, by defining $x \mapsto_{\mathsf{id}_x} x$ and $x \mapsto_{\mathsf{sub}_{x\setminus v}} v$.

Implications from left to right, by mutual induction. Consider a step $\xrightarrow{\rho}$.

- Case $t\ u \xrightarrow{\rho} t'\ u$ by rule APP-LEFT with $t \xrightarrow{\rho} t'$. By induction hypothesis, there are an evaluation context $\mathcal{E}(\bullet)$ and two terms r and r', such that $t = \mathcal{E}(r)$, $r \mapsto_\rho r'$, and $t' = \mathcal{E}(r')$. We conclude using the context $\mathcal{E}' = \mathcal{E}(\bullet)\ u$, which is an evaluation context.
- Case $t[x\backslash u] \xrightarrow{\rho} t'[x\backslash u]$ by rule ES-LEFT is identical.
- Case $t[x\backslash u] \xrightarrow{\rho} t[x\backslash u']$ by rule ES-RIGHT with $t \xrightarrow{\mathsf{id}_x} t$ and $u \xrightarrow{\rho} u'$. By induction hypotheses, we have $\mathcal{E}_1$ such that $t = \mathcal{E}_1(\!|x|\!)$ and $\mathcal{E}_2$, r, and r', such that $u = \mathcal{E}_2(r)$, $r \mapsto_\rho r'$, and $u' = \mathcal{E}_2(r')$. We conclude using the context $\mathcal{E}' = \mathcal{E}_1(\!|x|\!)[x\backslash\mathcal{E}_2(\bullet)]$, which is an evaluation context.
- Cases $x \xrightarrow{\mathsf{id}_x} x$ and $x \xrightarrow{\mathsf{sub}_{x\setminus v}} v$ by rules ID and SUB are immediate with the empty context.
- Case $t \xrightarrow{\mathsf{dB}} t'$ by rule DB with $t \to_{\mathsf{dB}} t'$ follows from Lemma 3.2 item 1.
- Case $t \xrightarrow{\mathsf{lsv}} t'$ by rule LSV with $t \to_{\mathsf{lsv}} t'$ follows from Lemma 3.2 item 2, by mutual induction.

Consider a step $\to_{\mathsf{lsv}}$.

- Case $t[x\backslash v] \to_{\mathsf{lsv}} t'[x\backslash v]$ by rule LSV-BASE with $t \xrightarrow{\mathsf{sub}_{x\setminus v}} t'$ and v value. By (mutual) induction hypothesis, there is $\mathcal{E}$ such that $t = \mathcal{E}(\!|x|\!)$ and $t' = \mathcal{E}(\!|v|\!)$. Then we have $\mathcal{E}(\!|x|\!)[x\backslash v] \mapsto_{\mathsf{lsv}} \mathcal{E}(\!|v|\!)[x\backslash v]$.
- Case $t[x\backslash u[y\backslash w]] \to_{\mathsf{lsv}} t'[y\backslash w]$ by rule LSV-σ with $t[x\backslash u] \to_{\mathsf{lsv}} t'$. By induction hypothesis, $t[x\backslash u] \mapsto_{\mathsf{lsv}} t'$. Then necessarily there are $\mathcal{E}$, v, and $\mathcal{L}$, with v a value, such that $t = \mathcal{E}(\!|x|\!)$, $u = v\mathcal{L}$, and $t' = \mathcal{E}(\!|v|\!)[x\backslash v]\mathcal{L}$. Then $t[x\backslash u[y\backslash w]] = \mathcal{E}(\!|x|\!)[x\backslash v\mathcal{L}[y\backslash w]] \mapsto_{\mathsf{lsv}} \mathcal{E}(\!|v|\!)[x\backslash v]\mathcal{L}[y\backslash w] = t'[y\backslash w]$.

Implications from right to left, by mutual induction. Consider a context and a reduction $r \mapsto_\rho r'$.

- Case $\mathcal{E} = \bullet$. By case on ρ.
  - If $r \mapsto_{\mathsf{dB}} r'$ then $\mathcal{E}(r) = r \xrightarrow{\mathsf{dB}} r' = \mathcal{E}(r')$ by Lemma 3.2 item 1.
  - If $r \mapsto_{\mathsf{lsv}} r'$ then $\mathcal{E}(r) = r \xrightarrow{\mathsf{lsv}} r' = \mathcal{E}(r')$ by mutual induction hypothesis.
  - If $r \mapsto_{\mathsf{id}_x} r'$ or $r \mapsto_{\mathsf{sub}_{x\setminus v}} r'$ then conclusion is by a base rule of $\xrightarrow{\rho}$.
- Case $\mathcal{E}(\bullet)\ u$. By induction hypothesis, $\mathcal{E}(r) \xrightarrow{\rho} \mathcal{E}(r')$, and then by rule APP-LEFT, $\mathcal{E}(r)\ u \xrightarrow{\rho} \mathcal{E}(r')\ u$.
- Case $\mathcal{E}(\bullet)[x\backslash u]$ is identical.
- Case $\mathcal{E}_1(\!|x|\!)[x\backslash\mathcal{E}_2(\bullet)]$. By induction hypotheses, $\mathcal{E}_1(\!|x|\!) \xrightarrow{\mathsf{id}_x} \mathcal{E}_1(\!|x|\!)$ and $\mathcal{E}_2(r) \xrightarrow{\rho} \mathcal{E}_2(r')$, and we conclude by rule ES-RIGHT.

Consider a step $\mapsto_{\mathsf{lsv}}$.

- Case $\mathcal{E}(\!|x|\!)[x\backslash v] \mapsto_{\mathsf{lsv}} \mathcal{E}(\!|v|\!)[x\backslash v]$. Conclusion is immediate by mutual induction hypothesis.
- Case $\mathcal{E}(\!|x|\!)[x\backslash v\mathcal{L}[y\backslash w]] \mapsto_{\mathsf{lsv}} \mathcal{E}(\!|v|\!)[x\backslash v]\mathcal{L}[y\backslash w]$. By induction hypothesis, $\mathcal{E}(\!|x|\!)[x\backslash v\mathcal{L}] \to_{\mathsf{lsv}} \mathcal{E}(\!|v|\!)[x\backslash v]\mathcal{L}$, and we conclude by rule LSV-σ.

□

4. Strong call-by-need calculus λ_{SN}

Our strong call-by-need calculus is defined by the same terms and reduction rules as λ_{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.

4.1. **Reduction in λ_{sn}.** The main reduction relation is written $t \to_{\mathsf{sn}} t'$ and represents one step of dB- or lsv-reduction at an eligible position of the term t. The starting point is the inductive definition of weak call-by-need reduction, which we now extend to account for strong reduction.

Top-level-like positions and frozen variables. Strong reduction brings new behaviors that cannot be observed in weak 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 [BBBK17], 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 $\lambda x.x\ (\lambda y.t')\ u$ (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 $(\lambda x.x\ (\lambda y.t')\ u)\ v$, *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 $(\lambda x.x\ (\lambda y.t')\ u)\ v$. Thus our calculus shall prevent them from being reduced until it gathers more information about v.

Therefore, the positions of a subterm t that are eligible for reduction largely depend on the context surrounding t. They depend in particular on the set of variables that are frozen by this context, which is itself mutually dependent with the notion of top-level-like positions. To characterize top-level-like positions and frozen variables, we use a judgment $\mathcal{C}(\bullet) \vdash \mu, \varphi$ where $\mathcal{C}(\bullet)$ is a context, μ is a boolean flag whose value can be $\top$ or $\bot$, and φ is a set of variables. A context hole $\mathcal{C}(\bullet)$ is in a top-level-like position if one can derive $\mathcal{C}(\bullet) \vdash \top, \varphi$. A variable x is frozen in the context $\mathcal{C}(\bullet)$ if one can derive $\mathcal{C}(\bullet) \vdash \mu, \varphi$ with $x \in \varphi$.

Figure 4 gives an inductive definition of the judgment $\mathcal{C}(\bullet) \vdash \mu, \varphi$. A critical aspect of this definition is the direction of the flow of information. Indeed, we deduce information on the hole of the context from the context itself. That is, information is flowing outside-in: from top-level toward the hole of the context.

Rule TOP-LEVEL states that the top-level position of a term is top-level-like, and that any (free) variable can be considered to be frozen at this point.

$$\textsc{top-level}\ \frac{}{\bullet \vdash \top, \varphi} \qquad \textsc{abs-}\top\ \frac{\mathcal{C}(\bullet) \vdash \top, \varphi}{\mathcal{C}(\lambda x.\bullet) \vdash \top, \varphi \cup \{x\}} \qquad \textsc{abs-}\bot\ \frac{\mathcal{C}(\bullet) \vdash \bot, \varphi}{\mathcal{C}(\lambda x.\bullet) \vdash \bot, \varphi}$$

$$\textsc{app-left}\ \frac{\mathcal{C}(\bullet) \vdash \mu, \varphi}{\mathcal{C}(\bullet\ t) \vdash \bot, \varphi} \qquad \textsc{app-right}\ \frac{\mathcal{C}(\bullet) \vdash \mu, \varphi}{\mathcal{C}(t\ \bullet) \vdash \bot, \varphi} \qquad \textsc{app-right-}\mathcal{S}\ \frac{\mathcal{C}(\bullet) \vdash \mu, \varphi \qquad t \in \mathcal{S}_\varphi}{\mathcal{C}(t\ \bullet) \vdash \top, \varphi}$$

$$\textsc{es-left}\ \frac{\mathcal{C}(\bullet) \vdash \mu, \varphi}{\mathcal{C}(\bullet[x\backslash u]) \vdash \mu, \varphi} \qquad \textsc{es-right}\ \frac{\mathcal{C}(\bullet) \vdash \mu, \varphi}{\mathcal{C}(t[x\backslash\bullet]) \vdash \bot, \varphi} \qquad \textsc{es-left-}\mathcal{S}\ \frac{\mathcal{C}(\bullet) \vdash \mu, \varphi \qquad u \in \mathcal{S}_\varphi}{\mathcal{C}(\bullet[x\backslash u]) \vdash \mu, \varphi \cup \{x\}}$$

Figure 4: Top-level-like positions and frozen variables.

$$\frac{x \in \varphi}{x \in \mathcal{S}_\varphi} \qquad \frac{t \in \mathcal{S}_\varphi}{t\ u \in \mathcal{S}_\varphi} \qquad \frac{t \in \mathcal{S}_\varphi}{t[x\backslash u] \in \mathcal{S}_\varphi} \qquad \frac{t \in \mathcal{S}_{\varphi \cup \{x\}} \qquad u \in \mathcal{S}_\varphi}{t[x\backslash u] \in \mathcal{S}_\varphi}$$

Figure 5: Structures of λ_{sn}.

Rules ABS-$\top$ and ABS-$\bot$ indicate that a variable x bound by an abstraction is considered to be frozen if this abstraction is at a top-level-like position. Moreover, if an abstraction is at a top-level-like position, then the position of its body is also top-level-like.

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 rules ABS-$\top$ and ABS-$\bot$ implicitly require that the variable x bound by the abstraction is fresh and hence *not* in the set φ. We keep this *freshness convention* in all the definitions of the paper. In particular, it also applies to the three rules dealing with explicit substitution.

Rules APP-LEFT, APP-RIGHT, ES-LEFT, and ES-RIGHT state that the variables frozen on either side of an application or an explicit substitution are the variables that are frozen at the level of the application or explicit substitution itself. Rule ES-LEFT indicates in addition that the top-level-like status of a position is inherited on the left side of an explicit substitution.

Rule APP-RIGHT-$\mathcal{S}$ strengthens the rule APP-RIGHT in the specific case where the application is led by a frozen variable. When this criterion is met, the argument position of an application can be considered as top-level-like. Rule ES-LEFT-$\mathcal{S}$ strengthens the rule ES-LEFT when the argument of an explicit substitution is led by a frozen variable. In this case, the variable of the substitution can itself be considered as frozen.

The specific criterion of the last two rules is made formal through 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 (Figure 5). We write $\mathcal{S}_\varphi$ the set of structures under a set φ of frozen variables. It differs from the notion in [BBBK17] in that it does not require the term to be in normal form. For example, we have $x\ t \in \mathcal{S}_{\{x\}}$ even if t still contains some redexes.

The reason for considering structures in rules APP-RIGHT-$\mathcal{S}$ and ES-LEFT-$\mathcal{S}$ is that they cannot reduce to an abstraction, even under an explicit substitution. For example, $x[x\backslash\lambda z.t]$ is not a structure, whatever the set φ. See also Lemma 5.3 for another take on structures.

Notice that, by our freshness condition, it is assumed that x is *not* in the set φ in the third and fourth rules of Figure 5. As implied by the last rule in Figure 5, an explicit

substitution in a structure may even affect the leading variable, provided that the content of the substitution is itself a structure, *e.g.*, $(x\ t)[x\backslash y\ u] \in \mathcal{S}_{\{y\}}$.

Example. Consider the term $\lambda x.((\lambda z.x\ z\ (\lambda y.y\ t)\ (z\ u))\ v)$, where t, u, and v are arbitrary, not necessarily closed, subterms. The above rules allow us to derive that the subterm $x\ z\ (\lambda y.y\ t)\ (z\ u)$ is at a position where x is a frozen variable.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bullet \vdash \top, \emptyset}\ \textsc{top-level}}{\lambda x.\bullet \vdash \top, \{x\}}\ \textsc{abs-}\top}{\lambda x.(\bullet\ v) \vdash \bot, \{x\}}\ \textsc{app-left}}{\lambda x.((\lambda z.\bullet)\ v) \vdash \bot, \{x\}}\ \textsc{abs-}\bot$$

This position, however, is *not* top-level-like, as it is the position of the body of an applied λ-abstraction. This can be observed in the previous derivation by the unavoidable application of the rule APP-LEFT. Independently of the fact that its position is not top-level-like, the subterm $x\ z\ (\lambda y.y\ t)\ (z\ u)$ is a structure, as it is led by the frozen variable x. We can then build a follow-up of this derivation, first deriving with a combination of the rules APP-LEFT and APP-RIGHT-$\mathcal{S}$ that the second argument $\lambda y.y\ t$ of this structure *is* at a top-level-like position, then deducing that y is frozen and that the subterm t is at a top-level-like position.

$$\textsc{app-right-}\mathcal{S}\ \dfrac{\textsc{abs-}\top\ \dfrac{\textsc{app-right-}\mathcal{S}\ \dfrac{\textsc{app-left}\ \dfrac{\lambda x.((\lambda z.\bullet)\ v) \vdash \bot, \{x\}}{\lambda x.((\lambda z.\bullet\ \ (z\ u))\ v) \vdash \bot, \{x\}} \qquad x\ z \in \mathcal{S}_{\{x\}}}{\lambda x.((\lambda z.(x\ z\ \bullet)\ (z\ u))\ v) \vdash \top, \{x\}}}{\lambda x.((\lambda z.x\ z\ (\lambda y.\bullet)\ (z\ u))\ v) \vdash \top, \{x, y\}} \qquad y \in \mathcal{S}_{\{x,y\}}}{\lambda x.((\lambda z.x\ z\ (\lambda y.y\ \bullet)\ (z\ u))\ v) \vdash \top, \{x, y\}}$$

Conversely, remark that the subterm $z\ u$, which it is at a top-level-like position, is not a structure since z is not frozen. Thus, u is not at a top-level-like position.

Inductive definition of strong call-by-need reduction. The definition of reduction in our strong call-by-need calculus extends the inductive definition given in Section 3. First of all, reductions have to be allowed, at least in some circumstances, inside λ-abstractions and on the right side of an application. The criteria we apply are:

- reduction under a needed λ-abstraction is always allowed, and
- reduction on the right side of an application is allowed if and only if this application is both needed and a structure (*i.e.,* led by a frozen variable).

Applying the latter criterion requires identifying the frozen variables, and this identification in turn requires keeping track of the top-level-like positions. For this, we define an inductive reduction relation $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'$ parameterized by a reduction rule ρ and by some context information φ and μ. This relation plays two roles: identifying a position where a reduction rule can be applied in t, and performing said reduction. The information on the context of the subterm t mirrors exactly the information given by a judgment $\mathcal{C}(\bullet) \vdash \mu, \varphi$. We thus have:

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

Consequently, the top-level reduction relation $t \to_{\mathsf{sn}} t'$ holds whenever t reduces to t' by rule dB or lsv in the empty context. In other words, $t \xrightarrow{\mathsf{dB},\varphi,\mu}_{\mathsf{sn}} t'$ or $t \xrightarrow{\mathsf{lsv},\varphi,\mu}_{\mathsf{sn}} t'$, where the flag μ is $\top$, and the set φ is typically empty when t is closed, or contains the free variables of t otherwise.

The inference rules for $\xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}}$ are given in Figure 6. Notice that ρ flows inside-out, *i.e.*, from the position of the reduction itself to top-level, as it was already the case in the definition of weak call-by-need reduction (Figure 2). On the contrary, information about φ and μ flows outside-in, that is from top-level to the position of the reduction. This also reflects the information flow already seen in the definition of the judgment $\mathcal{C}(\bullet) \vdash \mu, \varphi$. However, this now appears has an *upward* flow of information in the inference rules, which may seem like a reversal. This comes from the fact that the inference rules in Figure 6 focus on terms, while Figure 4 focused on contexts. Notice also that we do not rely explicitly on the judgment $\mathcal{C}(\bullet) \vdash \mu, \varphi$ in the inference rules. Instead, we blend the inference rules defining this judgment into the reduction rules.

Rule APP-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 APP-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. The latter is expressed using the notion of structure defined above (Figure 5).

Rules ABS-$\top$ and ABS-$\bot$ make reduction always possible inside a λ-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 ES-LEFT and ES-LEFT-$\mathcal{S}$ 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 ES-RIGHT restricts reduction inside a substitution to the case where an occurrence of the substituted variable is at a reducible position. It uses the auxiliary rule id_x already discussed in Section 3, which propagates using the same inductive reduction relation, to probe a term for the presence of some variable x at a reducible position. By freshness, in all the rules with a binder, the bound variable x can appear neither in φ nor in ρ, that is, if $\rho = \mathsf{id}_y$ then $x \neq y$, and if $\rho = \mathsf{sub}_{y \setminus v}$ then $x \neq y$ and $x \notin \mathsf{fv}(v)$.

Rules DB and LSV are the base cases for applying reductions dB or lsv. As in Section 3, each is defined using an auxiliary reduction relation dealing with the list $\mathcal{L}$ of explicit substitutions. These auxiliary reductions are given in Figure 7. They differ from the ones from Figure 3 in two ways.

First, since the inference rule LSV-BASE uses as a premise a reduction $t \xrightarrow{\mathsf{sub}_{x \setminus v},\varphi,\mu}_{\mathsf{sn}} t'$, the auxiliary relation $\xrightarrow{\varphi,\mu}_{\mathsf{lsv}}$ has to keep track of the context information. It is thus parameterized by φ and μ. The combination of these parameters and of the rules LSV and LSV-BASE makes it possible, in the case of a 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 [BBBK17], a similar effect was achieved using a more convoluted condition on a composition of contexts.

Second, a new rule LSV-σ-$\mathcal{S}$ appears, which strengthens the already discussed LSV-σ rule in the same way that ES-LEFT-$\mathcal{S}$ strengthens ES-LEFT. The difference between LSV-σ and LSV-σ-$\mathcal{S}$ can be ignored until Section 5.

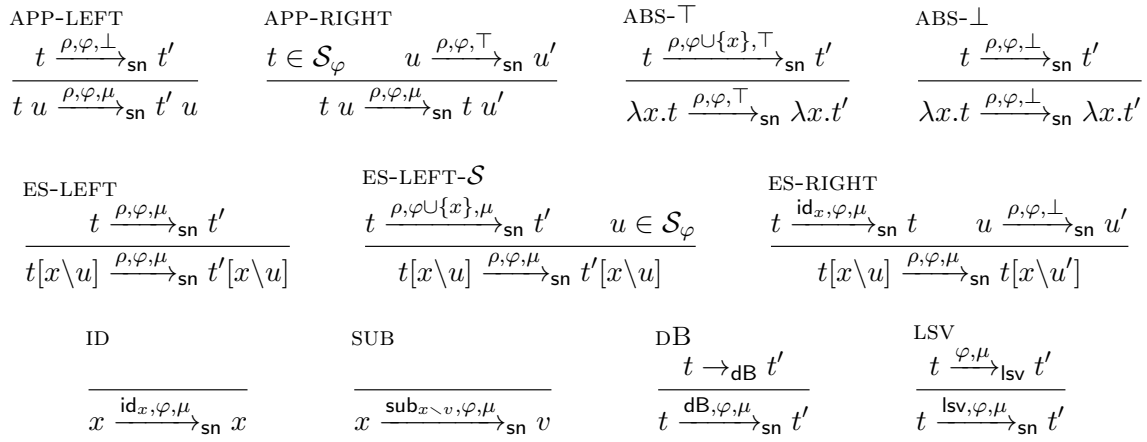


Figure 6: Reduction rules for λ_{sn}.

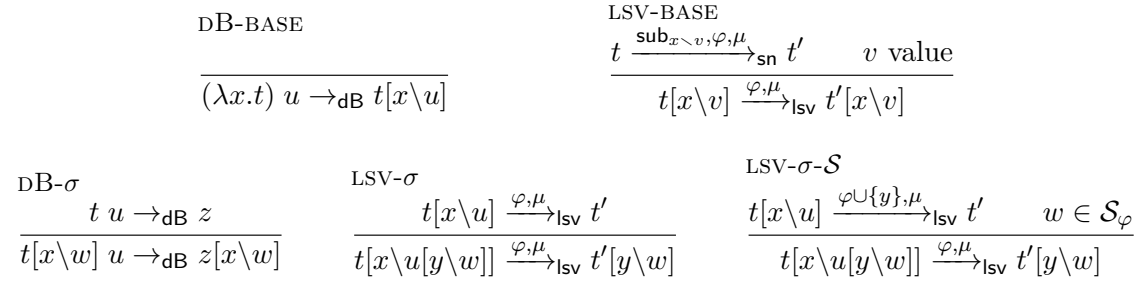
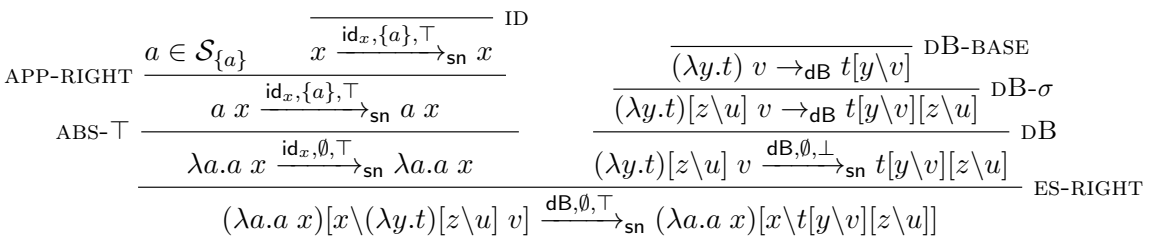


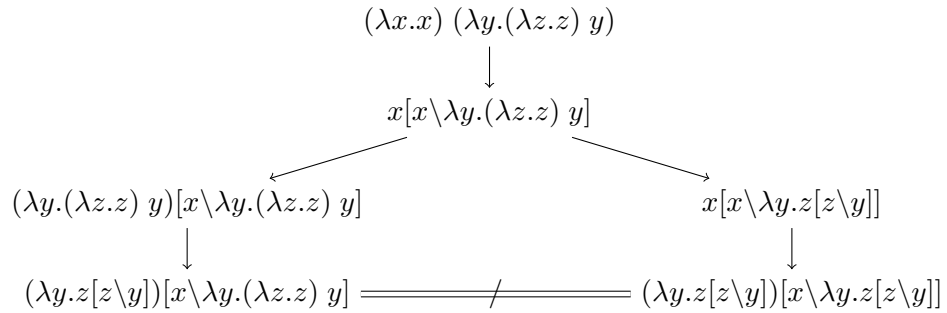
Figure 7: Auxiliary reduction rules for λ_{sn}.

Example. The reduction $(\lambda a.a\ x)[x\backslash(\lambda y.t)[z\backslash u]\ v] \to_{\mathsf{sn}} (\lambda a.a\ x)[x\backslash t[y\backslash v][z\backslash u]]$ is allowed by λ_{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.



This example also shows that top-level-like positions and evaluation positions are different sets, even if their definitions are related. Indeed, we can observe that the position actually reduced by the rule DB is labeled by $\bot$. On the other hand, the definition of top-level-like positions in Figure 4 may label as top-level-like a position that would not be considered for reduction, as the position of t in the term $\lambda x.((\lambda y.y\ (x\ t))\ u)$, which is an argument position in a structure (led by the frozen variable x) that is itself in argument position of another application, the latter not being known to be a structure.

Basic properties. The calculus λ_{sn} as we just defined it is not confluent. This can be observed in the reduction of the term $(\lambda x.x)\ (\lambda y.(\lambda z.z)\ y)$ below.



After the first reduction step, we arrive at the term $x[x\backslash\lambda y.(\lambda z.z)\ y]$ where the only occurrence of x is at an evaluation position, and two reductions are possible: substituting the term $\lambda y.(\lambda z.z)\ y$, which is a value, or reducing the redex $(\lambda z.z)\ y$ inside this value. The latter case, on the right path, leads to the normal form $(\lambda y.z[z\backslash y])[x\backslash\lambda y.z[z\backslash y]]$ after reducing the only remaining redex. The former case, on the left path, can only lead to $(\lambda y.z[z\backslash y])[x\backslash\lambda y.(\lambda z.z)\ y]$, where the redex $\lambda y.(\lambda z.z)\ y$ in the explicit substitution will never be accessed again since the variable x does not appear in the term anymore. However, as a corollary of Theorem 4.4, we know that such a defect of confluence may only happen inside an unreachable substitution. Moreover, we will see in Section 5 that strict confluence can be restored in a restriction of the calculus.

Finally, note that the strong call-by-need strategy introduced in [BBBK17] is included in our calculus. One can recover this strategy by imposing two restrictions on $\xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}}$:

- remove the rule ABS-$\bot$, so as to only reduce abstractions that are in top-level-like positions;
- restrict the rule APP-RIGHT 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.

4.2. **Soundness.** The calculus λ_{sn} is sound with respect to the λ-calculus, in the sense that any normalizing reduction in λ_{sn} can be related to a normalizing β-reduction through unfolding. This section establishes this result (Theorem 4.4). All the proofs in this section are formalized in Abella (see Section 6).

The first part of the proof requires relating λ_{sn}-reduction to β-reduction.

Lemma 4.1 (Simulation)**.** *If* $t \to_{\mathsf{sn}} t'$ *then* $t^\star \to_\beta^* t'^\star$.

Proof. By induction on the reduction $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'$. □

The second part requires relating the normal forms of λ_{sn} to β-normal forms. The normal forms of λ_{sn} correspond to the normal forms of the strong call-by-need strategy [BBBK17]. They can be characterized by the inductive definition given in Figure 8. Please note a subtlety of this definition: there are two rules for terms of the form $t[x\backslash u]$. One applies only when u is a structure, and allows freezing the variable x in t. The other applies without any constraint on u, but requires t to be a normal form without x being frozen (by our freshness convention on variables, φ cannot contain x). Since variables are considered normal forms only when frozen, the latter case requires x not to appear in any position of t whose normality is checked. In fact, for this rule to apply successfully, x may appear only in

$$\frac{x \in \varphi}{x \in \mathcal{N}_\varphi} \qquad \frac{t \in \mathcal{N}_\varphi \qquad t \in \mathcal{S}_\varphi \qquad u \in \mathcal{N}_\varphi}{t\ u \in \mathcal{N}_\varphi} \qquad \frac{t \in \mathcal{N}_{\varphi \cup \{x\}}}{\lambda x.t \in \mathcal{N}_\varphi}$$

$$\frac{t \in \mathcal{N}_{\varphi \cup \{x\}} \qquad u \in \mathcal{N}_\varphi \qquad u \in \mathcal{S}_\varphi}{t[x\backslash u] \in \mathcal{N}_\varphi} \qquad \frac{t \in \mathcal{N}_\varphi}{t[x\backslash u] \in \mathcal{N}_\varphi}$$

Figure 8: Normal forms of λ_{sn}.

positions of t that can be removed by one or more steps of gc. Hence, $(\lambda a.a\ x)[x\backslash r]$ is not a normal form if r is a redex, but $(\lambda a.a\ x)[y\backslash z][z\backslash r]$ is, for any term r.

Lemma 4.2 (Normal forms)**.** *$t \in \mathcal{N}_\varphi$ if and only if there is no reduction $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'$.*

Proof. The first part (a term cannot be both 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. □

Please note that the characterization of normal forms in Figure 8 and the associated Lemma 4.2 take into account the fact that we did not include any gc rule in the calculus. If that were the case, the characterization of normal forms could be adapted, by removing the fifth rule.

Lemma 4.3 (Unfolded normal forms)**.** *If $t \in \mathcal{N}_\varphi$ then $t^\star$ is a normal form in the λ-calculus.*

Proof. By induction on $t \in \mathcal{N}_\varphi$. □

Soundness is a direct consequence of the three previous lemmas.

Theorem 4.4 (Soundness)**.** *Let t be a λ_{sn}-term. If there is a reduction $t \to^*_{\mathsf{sn}} u$ with u a λ_{sn}-normal form, then $u^\star$ is the β-normal form of $t^\star$.*

This theorem implies that all the λ_{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 $(\lambda y.z[z\backslash y])[x\backslash \lambda y.(\lambda z.z)\ y]$ and $(\lambda y.z[z\backslash y])[x\backslash \lambda y.z[z\backslash y]]$, but both of them unfold to $\lambda y.y$.

4.3. **Completeness.** Our strong call-by-need calculus is complete with respect to normalization in the λ-calculus in a strong sense: Whenever a λ-term t admits a normal form in the pure λ-calculus, every reduction path in λ_{sn} eventually reaches a representative of this normal form. This section is devoted to proving this completeness result (Theorem 4.10). The proof relies on the non-idempotent intersection type system in the following way. First, typability (Theorem 2.1) ensures that any weakly normalizing λ-term admits a typing derivation (with no positive occurrence of $\{\!\{\}\!\}$). Second, we prove that any λ_{sn}-reduction in a typed λ_{sn}-term t (with no positive occurrence of $\{\!\{\}\!\}$) is at a typed position of t (Theorem 4.9). Third, weighted subject reduction (Theorem 2.2) provides a decreasing measure for λ_{sn}-reduction. Finally, the obtained normal form is related to the β-normal form using Lemmas 4.1, 4.2, and 4.3.

The proof of the forthcoming typed reduction (Theorem 4.9) uses a refinement of the non-idempotent intersection types system of λ_{c}, given in Figure 9. Both systems derive the same typing judgments with the same typed positions. The refined system, however, features an annotated typing judgment $\Gamma \vdash^\mu_\varphi t : \tau$ embedding the same context information

$$\textsc{ty-var}\ \frac{}{x : \{\!\{\sigma\}\!\} \vdash^{\mu}_{\varphi} x : \sigma} \qquad \textsc{ty-abs-}\bot\ \frac{\Gamma, x : \mathcal{M} \vdash^{\bot}_{\varphi} t : \tau}{\Gamma \vdash^{\bot}_{\varphi} \lambda x.t : \mathcal{M} \to \tau} \qquad \textsc{ty-abs-}\top\ \frac{\Gamma, x : \mathcal{M} \vdash^{\top}_{\varphi \cup \{x\}} t : \tau}{\Gamma \vdash^{\top}_{\varphi} \lambda x.t : \mathcal{M} \to \tau}$$

$$\textsc{ty-app}\ \frac{\Gamma \vdash^{\bot}_{\varphi} t : \mathcal{M} \to \tau \qquad (\Delta_\sigma \vdash^{\bot}_{\varphi} u : \sigma)_{\sigma \in \mathcal{M}}}{\Gamma + \sum_{\sigma \in \mathcal{M}} \Delta_\sigma \vdash^{\mu}_{\varphi} t\ u : \tau} \qquad \textsc{ty-app-}\mathcal{S}\ \frac{\Gamma \vdash^{\bot}_{\varphi} t : \mathcal{M} \to \tau \quad t \in \mathcal{S}_\varphi \quad (\Delta_\sigma \vdash^{\top}_{\varphi} u : \sigma)_{\sigma \in \mathcal{M}}}{\Gamma + \sum_{\sigma \in \mathcal{M}} \Delta_\sigma \vdash^{\mu}_{\varphi} t\ u : \tau}$$

$$\textsc{ty-es}\ \frac{\Gamma, x : \mathcal{M} \vdash^{\mu}_{\varphi} t : \tau \quad (\Delta_\sigma \vdash^{\bot}_{\varphi} u : \sigma)_{\sigma \in \mathcal{M}}}{\Gamma + \sum_{\sigma \in \mathcal{M}} \Delta_\sigma \vdash^{\mu}_{\varphi} t[x \backslash u] : \tau} \qquad \textsc{ty-es-}\mathcal{S}\ \frac{\Gamma, x : \mathcal{M} \vdash^{\mu}_{\varphi \cup \{x\}} t : \tau \quad u \in \mathcal{S}_\varphi \quad (\Delta_\sigma \vdash^{\bot}_{\varphi} u : \sigma)_{\sigma \in \mathcal{M}}}{\Gamma + \sum_{\sigma \in \mathcal{M}} \Delta_\sigma \vdash^{\mu}_{\varphi} t[x \backslash u] : \tau}$$

Figure 9: Annotated system of non-idempotent intersection types.

that are defined by the judgment $\mathcal{C}(\bullet) \vdash \mu, \varphi$ and used in the inductive reduction relation $\xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}}$, namely the set φ of frozen variables at the considered position and a marker μ of top-level-like positions. These annotations are faithful counterparts to the corresponding annotations of λ_{sn} 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 TY-ABS-$\bot$ and TY-ABS-$\top$, 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 into two versions: TY-APP-$\mathcal{S}$ is applicable when the left part of the application is a structure and marks the right part as a $\top$ position, while TY-APP 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 $\bot$ position. Finally, the rule for typing an explicit substitution is similarly split into 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 $\bot$ position, since this position is never top-level-like.

As it was the case for evaluation positions, the definition of typed positions largely depends on the notions of frozen variables and top-level-like positions, but the sets of typed positions and of top-level-like positions do not have any particular relation to each other. Some typed positions are not top-level-like (*e.g.*, on the left of a typed application), and some top-level-like positions are not typed (*e.g.*, the argument of an untyped structure). On the other hand, there is an actual connection between the set of typed positions and the set of evaluation positions, given by Theorem 4.9.

We write $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t : \tau$ if there is a derivation Φ of the annotated typing judgment $\Gamma \vdash^{\mu}_{\varphi} t : \tau$. We denote by $\mathsf{fzt}(\Phi)$ the set of types associated to frozen variables in judgments of the derivation Φ.

Lemma 4.5 (Typing derivation annotation)**.** *If there is a derivation $\Phi \rhd \Gamma \vdash t : \tau$, then for any φ and μ there is a derivation $\Phi' \rhd \Gamma \vdash^{\mu}_{\varphi} t : \tau$ such that the sets of typed positions in Φ and Φ' are equal.*

Proof. By induction on Φ, since annotations do not interfere with typing. □

The converse property is also true, by erasure of the annotations, but it is not used in the proof of the completeness result.

The most crucial part of the proof of Theorem 4.9 is ensuring that any argument of a typed structure is itself at a typed position. This follows from the following three lemmas.

Lemma 4.6 (Typed structure)**.** *If $\Gamma \vdash^{\mu}_{\varphi} t : \tau$ and $t \in \mathcal{S}_{\varphi}$, then there is $x \in \varphi$ such that $\tau \in \mathcal{T}_{+}(\Gamma(x))$.*

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 τ. Let us give some more details.

- Case $t = x$. By inversion of $x \in \mathcal{S}_{\varphi}$ we deduce $x \in \varphi$. Moreover the only rule applicable to derive $\Gamma \vdash^{\mu}_{\varphi} x : \tau$ is TY-VAR, which gives the conclusion.
- Case $t = t_1\ t_2$. By inversion of $t_1\ t_2 \in \mathcal{S}_{\varphi}$ we deduce $t_1 \in \mathcal{S}_{\varphi}$. Moreover the only rules applicable to derive $\Gamma \vdash^{\mu}_{\varphi} t_1\ t_2 : \tau$ are TY-APP and TY-APP-$\mathcal{S}$. Both have a premise $\Gamma' \vdash^{\perp}_{\varphi} t_1 : \mathcal{M} \to \tau$ with $\Gamma' \subseteq \Gamma$, to which the induction hypothesis applies, ensuring $\mathcal{M} \to \tau \in \mathcal{T}_{+}(\Gamma'(x))$ and thus $\tau \in \mathcal{T}_{+}(\Gamma'(x))$ and $\tau \in \mathcal{T}_{+}(\Gamma(x))$.
- Case $t = t_1[x\backslash t_2]$. We reason by case on the last rules applied to derive $t_1[x\backslash t_2] \in \mathcal{S}_{\varphi}$ and $\Gamma \vdash^{\mu}_{\varphi} t_1[x\backslash t_2] : \tau$. There are two possible rules for each.
  - The case $t_1[x\backslash t_2] \in \mathcal{S}_{\varphi}$ is deduced from $t_1 \in \mathcal{S}_{\varphi}$ (with $x \notin \varphi$), and $\Gamma \vdash^{\mu}_{\varphi} t_1[x\backslash t_2] : \tau$ comes from rule TY-ES. This rule has in particular a premise $\Gamma' \vdash^{\mu}_{\varphi} t_1 : \tau$ for a $\Gamma' = \Gamma'', x : \mathcal{M}$ such that $\Gamma'' \subseteq \Gamma$. We thus have by induction hypothesis on t_1 that $\tau \in \mathcal{T}_{+}(\Gamma'(y))$ for some $y \in \varphi \cap \mathsf{dom}(\Gamma')$. Since $y \in \varphi$ and $x \notin \varphi$, we have $y \neq x$. Then, $y \in \mathsf{dom}(\Gamma'')$, $y \in \mathsf{dom}(\Gamma)$, and $\Gamma(y) = \Gamma''(y)$.
  - In the three other cases, we have:
    (1) a hypothesis $t_1 \in \mathcal{S}_{\varphi}$ or $t_1 \in \mathcal{S}_{\varphi \cup \{x\}}$, from which we deduce $t_1 \in \mathcal{S}_{\varphi \cup \{x\}}$,
    (2) a hypothesis $\Gamma' \vdash^{\mu}_{\varphi} t_1 : \tau$ or $\Gamma' \vdash^{\mu}_{\varphi \cup \{x\}} t_1 : \tau$ (for a $\Gamma' = \Gamma'', x : \mathcal{M}$ such that $\Gamma'' \subseteq \Gamma$), from which we deduce $\Gamma' \vdash^{\mu}_{\varphi \cup \{x\}} t_1 : \tau$, and
    (3) a hypothesis $t_2 \in \mathcal{S}_{\varphi}$, coming from the derivation of $t_1[x\backslash t_2]$ or the derivation of $\Gamma \vdash^{\mu}_{\varphi} t_1[x\backslash t_2] : \tau$ (or both).

    Then by induction hypothesis on t_1, we have $\tau \in \mathcal{T}_{+}(\Gamma'(y))$ for some $y \in \varphi \cup \{x\}$.
    * If $y \neq x$, then $y \in \varphi$ and $\Gamma(y) = \Gamma''(y)$, which allows a direct conclusion.
    * If $y = x$, then $\tau \in \mathcal{T}_{+}(\Gamma'(x))$ implies $\mathcal{M} \neq \{\!\{\}\!\}$. Let $\sigma \in \mathcal{M}$ with $\tau \in \mathcal{T}_{+}(\sigma)$. The instance of the rule TY-ES or TY-ES-$\mathcal{S}$ we consider thus has at least one premise $\Delta \vdash^{\perp}_{\varphi} t_2 : \sigma$ with $\Delta \subseteq \Gamma$. Since $t_2 \in \mathcal{S}_{\varphi}$, by induction hypothesis on t_2, there is $z \in \varphi \cap \mathsf{dom}(\Delta)$ such that $\sigma \in \mathcal{T}_{+}(\Delta(z))$. Then, $\tau \in \mathcal{T}_{+}(\Delta(z))$ and $\tau \in \Gamma$. □

Lemma 4.7 (Subformula property)**.**

(1) *If $\Phi \rhd \Gamma \vdash^{\top}_{\varphi} t : \tau$ then* $\begin{cases} \mathcal{T}_{+}(\mathit{fzt}(\Phi)) \subseteq \bigcup_{x \in \varphi} \mathcal{T}_{+}(\Gamma(x)) \cup \mathcal{T}_{-}(\tau) \\ \mathcal{T}_{-}(\mathit{fzt}(\Phi)) \subseteq \bigcup_{x \in \varphi} \mathcal{T}_{-}(\Gamma(x)) \cup \mathcal{T}_{+}(\tau) \end{cases}$

(2) *If $\Phi \rhd \Gamma \vdash^{\perp}_{\varphi} t : \tau$ then* $\begin{cases} \mathcal{T}_{+}(\mathit{fzt}(\Phi)) \subseteq \bigcup_{x \in \varphi} \mathcal{T}_{+}(\Gamma(x)) \\ \mathcal{T}_{-}(\mathit{fzt}(\Phi)) \subseteq \bigcup_{x \in \varphi} \mathcal{T}_{-}(\Gamma(x)) \end{cases}$

Proof. By mutual induction on the typing derivations. Most cases are fairly straightforward. The only difficult case comes from the rule TY-APP-$\mathcal{S}$, in which there is a premise $\Delta \vdash^{\top}_{\varphi} u : \sigma$ with mode $\top$ but with a type σ that does not clearly appear in the conclusion. Here we need the typed structure (Lemma 4.6) to conclude. Let us give some more details.

- Both properties are immediate in case TY-VAR, where $\mathsf{fzt}(\Phi) = \{\sigma\}$.
- Cases for abstractions.
  - If $\Phi \rhd \Gamma \vdash^{\perp}_{\varphi} \lambda x.t : \mathcal{M} \to \tau$ by rule TY-ABS-$\perp$ with premise $\Phi' \rhd \Gamma, x : \mathcal{M} \vdash^{\perp}_{\varphi} t : \tau$. Write $\Gamma' = \Gamma, x : \mathcal{M}$. By induction hypothesis we have $\mathcal{T}_+(\mathsf{fzt}(\Phi')) \subseteq \bigcup_{y\in\varphi} \mathcal{T}_+(\Gamma'(y))$. Since $x \notin \varphi$ by renaming convention, we deduce that $\mathcal{T}_+(\mathsf{fzt}(\Phi')) \subseteq \bigcup_{y\in\varphi} \mathcal{T}_+(\Gamma(y))$ and $\mathcal{T}_+(\mathsf{fzt}(\Phi)) \subseteq \bigcup_{y\in\varphi} \mathcal{T}_+(\Gamma(y))$. The same applies to negative type occurrences, which concludes the case.
  - If $\Phi \rhd \Gamma \vdash^{\top}_{\varphi} \lambda x.t : \mathcal{M} \to \tau$ by rule TY-ABS-$\top$ with premise $\Phi' \rhd \Gamma, x : \mathcal{M} \vdash^{\top}_{\varphi\cup\{x\}} t : \tau$. Write $\Gamma' = \Gamma, x : \mathcal{M}$. By induction hypothesis we have
$$\begin{array}{rcl} \mathcal{T}_+(\mathsf{fzt}(\Phi')) & \subseteq & \bigcup_{y\in(\varphi\cup\{x\})} \mathcal{T}_+(\Gamma'(y)) \cup \mathcal{T}_-(\tau) \\ & = & \bigcup_{y\in\varphi} \mathcal{T}_+(\Gamma(y)) \cup \mathcal{T}_+(\mathcal{M}) \cup \mathcal{T}_-(\tau) \\ & = & \bigcup_{y\in\varphi} \mathcal{T}_+(\Gamma(y)) \cup \mathcal{T}_-(\mathcal{M} \to \tau) \end{array}$$
    Thus $\mathcal{T}_+(\mathsf{fzt}(\Phi)) \subseteq \bigcup_{y\in\varphi} \mathcal{T}_+(\Gamma(y)) \cup \mathcal{T}_-(\mathcal{M} \to \tau)$. The same applies to negative occurrences, which concludes the case.
- Cases for application.
  - Cases for TY-APP are by immediate application of the induction hypotheses.
  - If $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t\ u : \tau$ by rule TY-APP-$\mathcal{S}$, with premises $\Phi_t \rhd \Gamma_t \vdash^{\perp}_{\varphi} t : \mathcal{M} \to \tau$, $t \in \mathcal{S}_\varphi$ and $\Phi_\sigma \rhd \Delta_\sigma \vdash^{\top}_{\varphi} u : \sigma$ for $\sigma \in \mathcal{M}$, with $\Gamma_t \subseteq \Gamma$ and $\Gamma_\sigma \subseteq \Gamma$ for all $\sigma \in \mathcal{M}$. Independently of the value of μ, we show that $\mathcal{T}_+(\mathsf{fzt}(\Phi)) \subseteq \bigcup_{x\in\varphi} \mathcal{T}_+(\Gamma(x))$ and $\mathcal{T}_-(\mathsf{fzt}(\Phi)) \subseteq \bigcup_{x\in\varphi} \mathcal{T}_-(\Gamma(x))$ to conclude on both sides of the mutual induction.
    Directly from the induction hypothesis, $\mathcal{T}_+(\mathsf{fzt}(\Phi_t)) \subseteq \bigcup_{x\in\varphi} \Gamma_t(x) \subseteq \mathcal{T}_+(\mathsf{fzt}(\Phi))$. By induction hypothesis on the other premises we have $\mathcal{T}_+(\mathsf{fzt}(\Phi_\sigma)) \subseteq \bigcup_{x\in\varphi} \Gamma_\sigma(x) \cup \mathcal{T}_-(\tau)$ for $\sigma \in \mathcal{M}$. We immediately have $\bigcup_{x\in\varphi} \Gamma_\sigma(x) \subseteq \bigcup_{x\in\varphi} \Gamma(x)$. We conclude by showing that $\mathcal{T}_-(\sigma) \subseteq \mathcal{T}_+(\Gamma_t(x))$ for some $x \in \varphi$. Since $t \in \mathcal{S}_\varphi$, by the first subformula property and the typing hypothesis on t we deduce that there is a $x \in \varphi$ such that $\mathcal{M} \to \tau \in \mathcal{T}_+(\Gamma_t(x))$. By closeness of type occurrences sets $\mathcal{T}_+(\tau)$ this means $\mathcal{T}_+(\mathcal{M} \to \tau) \subseteq \mathcal{T}_+(\Gamma_t(x))$. By definition $\mathcal{T}_+(\mathcal{M} \to \tau) = \mathcal{T}_-(\mathcal{M}) \cup \mathcal{T}_+(\tau) = \bigcup_{\sigma\in\mathcal{M}} \mathcal{T}_-(\sigma) \cup \mathcal{T}_+(\tau)$, which allows us to conclude the proof that $\bigcup_{x\in\varphi} \Gamma_\sigma(x) \cup \mathcal{T}_-(\tau) \subseteq \bigcup_{x\in\varphi} \Gamma(x)$.
    The same argument also applies to negative positions, and concludes the case.
- Cases for explicit substitution immediately follow the induction hypothesis. □

Lemma 4.8 (Typed structure argument)**.** *If $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t : \tau$ with $\{\!\{\}\!\} \notin \mathcal{T}_+(\Gamma \vdash t : \tau)$, then every typing judgment of the shape $\Gamma' \vdash^{\mu'}_{\varphi'} s : \mathcal{M} \to \sigma$ in Φ with $s \in \mathcal{S}_{\varphi'}$ satisfies $\mathcal{M} \neq \{\!\{\}\!\}$.*

Proof. Let $\Gamma' \vdash^{\mu'}_{\varphi'} s : \mathcal{M} \to \sigma$ in Φ with $s \in \mathcal{S}_{\varphi'}$. By Lemma 4.6, there is $x \in \varphi'$ such that $\mathcal{M} \to \sigma \in \mathcal{T}_+(\Gamma'(x))$. Then $\mathcal{M} \in \mathcal{T}_-(\Gamma'(x))$ and $\mathcal{M} \in \mathcal{T}_-(\mathsf{fzt}(\Phi))$. By Lemma 4.7, $\mathcal{M} \in \mathcal{T}_+(\Gamma \vdash^{\mu}_{\varphi} t : \tau)$, thus $\mathcal{M} \neq \{\!\{\}\!\}$. □

Theorem 4.9 (Typed reduction)**.** *If $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t : \tau$ with $\{\!\{\}\!\} \notin \mathcal{T}_+(\Gamma \vdash t : \tau)$, then every λ_{sn}-reduction $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'$ is at a typed position.*

Proof. We prove by induction on $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'$ that, if $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t : \tau$ with Φ such that any typing judgment $\Gamma' \vdash^{\mu'}_{\varphi'} s : \mathcal{M} \to \sigma$ in Φ with $s \in \mathcal{S}_{\varphi'}$ satisfies $\mathcal{M} \neq \{\!\{\}\!\}$, then $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'$ reduces at a typed position (the restriction on Φ is enabled by Lemma 4.8). Since all the other reduction cases concern positions that are systematically typed, we focus here on APP-RIGHT and ES-RIGHT.

- Case APP-RIGHT: $t\ u \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t\ u'$ with $t \in \mathcal{S}_\varphi$ and $u \xrightarrow{\rho,\varphi,\top}_{\mathsf{sn}} u'$, assuming $\Phi \rhd \Gamma \vdash^\mu_\varphi t\ u : \sigma$. By inversion of the last rule in Φ we know there is a subderivation $\Phi' \rhd \Gamma' \vdash^\bot_\varphi t : \mathcal{M} \to \sigma$ and by hypothesis $\mathcal{M} \neq \{\!\{\}\!\}$. Then u is typed in Φ and we can conclude by induction hypothesis.
- Case ES-RIGHT: $t[x\backslash u] \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t[x\backslash u']$ with $t \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}} t'$ and $u \xrightarrow{\rho,\varphi,\bot}_{\mathsf{sn}} u'$, assuming $\Phi \rhd \Gamma \vdash^\mu_\varphi t[x\backslash u] : \tau$. By inversion of the last rule in Φ we kown there is a subderivation $\Phi' \rhd \Gamma', x : \mathcal{M} \vdash^\mu_\varphi t : \tau$. By induction hypothesis we know that reduction $t \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}} t'$ is at a typed position in Φ', thus x is typed in t and $\mathcal{M} \neq \{\!\{\}\!\}$. Then u is typed in Φ and we can conclude by induction hypothesis on $u \xrightarrow{\rho,\varphi,\bot}_{\mathsf{sn}} u'$. □

Theorem 4.10 (Completeness)**.** *If a λ-term t is weakly normalizing in the λ-calculus, then t is strongly normalizing in λ_{sn}. Moreover, if n_β is the normal form of t in the λ-calculus, then any normal form n_{sn} of t in λ_{sn} is such that $n^\star_{\mathsf{sn}} = n_\beta$.*

Proof. Let t be a pure λ-term that admits a normal form n_β for β-reduction. By Theorem 2.1 there exists a derivable typing judgment $\Gamma \vdash t : \tau$ such that $\{\!\{\}\!\} \notin \mathcal{T}_+(\Gamma \vdash t : \tau)$. Thus by Theorems 4.9 and 2.2, the term t is strongly normalizing for $\to_{\mathsf{sn}}$. Let $t \to^*_{\mathsf{sn}} n_{\mathsf{sn}}$ be a maximal reduction in λ_{sn}. By Lemma 4.2, $n_{\mathsf{sn}} \in \mathcal{N}_\varphi$, and by Lemma 4.3, $n^\star_{\mathsf{sn}}$ is a normal form in the λ-calculus. Moreover, by simulation (Lemma 4.1), there is a reduction $t^\star \to^*_\beta n^\star_{\mathsf{sn}}$. By uniqueness of the normal form in the λ-calculus, $n^\star_{\mathsf{sn}} = n_\beta$. □

Note that, despite the fact that λ_{sn} does not enjoy the diamond property, our theorems of soundness (Theorem 4.4) and completeness (Theorem 4.10) imply that, in λ_{sn}, a term is weakly normalizing if and only if it is strongly normalizing.

These termination and completeness results also allow us to formalize our claim that λ_{sn} reduces only "needed" redexes.

Theorem 4.11 (Neededness)**.** *Consider a λ_{sn}-term t such that $t^\star$ is weakly normalizing in the λ-calculus. For any dB-reduction $t \xrightarrow{\mathsf{dB},\varphi,\mu}_{\mathsf{sn}} t'$ in λ_{sn}, the subterm r of t that is reduced is such that at least one occurrence of the corresponding β-redex r' in $t^\star$ is needed.*

Proof. Suppose no occurrence of the redex r' is needed in $t^\star$. Then by definition there is a β-reduction sequence of $t^\star$ that leads to the normal form of $t^\star$ without reducing any copy of r' nor any of its residuals (and the normal form, being normal, does not contain any residual of r'). Write p the position of r in t, and $t(\Omega)_p$ the term obtained from t by replacing r by the diverging term Ω. Then the term $(t(\Omega)_p)^\star$ is equal to the term obtained from $t^\star$ by replacing each copy of r' by Ω, and can still be β-normalized by the same β-reduction sequence. Thus, this term is weakly normalizing in the λ-calculus.

Since $(t(\Omega)_p)^\dagger$ β-reduces to $(t(\Omega)_p)^\star$, we deduce that the pure term $(t(\Omega)_p)^\dagger$ is also weakly normalizing in the λ-calculus. Thus by Theorem 2.1 there is a typing judgment $\Gamma \vdash (t(\Omega)_p)^\dagger : \tau$ such that $\{\!\{\}\!\} \notin \mathcal{T}_+(\Gamma \vdash t : \tau)$, and by Proposition 2.4 we also have $\Gamma \vdash t(\Omega)_p : \tau$. Then by Theorems 4.9 and 2.2, the term $t(\Omega)_p$ is strongly normalizing in λ_{sn}.

However, if $\to_{\mathsf{sn}}$ allows reducing r in t, then it also allows reducing Ω in $t(\Omega)_p$, as well as any reduct of Ω at the same position, which implies the existence of an infinite $\to_{\mathsf{sn}}$-reduction sequence from t: contradiction. Thus at least one occurrence of the β-redex r' is needed in $t^\star$. □

$$\textsc{lsv-base}\quad \frac{t \xrightarrow{\mathsf{sub}_{x\setminus v},\varphi,\mu}_{\mathsf{sn}} t' \qquad v \in \mathcal{N}_{\varphi,\emptyset,\bot} \qquad v \text{ value}}{t[x\backslash v] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t'[x\backslash v]}$$

Figure 10: New rule LSV-BASE for $\lambda_{\mathsf{sn}+}$.

5. Relatively optimal strategies

Our proposed λ_{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 $\lambda_{\mathsf{sn}+}$, has the same rules as λ_{sn} (Figures 6 and 7), except that LSV-BASE is replaced by the rule shown in Figure 10.

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 λ_{sn}. This makes it a relatively optimal strategy.

5.1. Local normal forms.

In λ_{c} and λ_{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 $\mathsf{id} = \lambda x.x$. The sequence of reductions does not depend on the set φ of frozen variables nor on the position μ, so we do not write them to lighten a bit the notations. Subterms that are about to be substituted or reduced are underlined.

$$\begin{aligned}
\underline{(\lambda w.w\ w)\ (\lambda y.\mathsf{id}\ y)} &\xrightarrow{\mathsf{db}}_{\mathsf{sn}} (\underline{w}\ w)[w\backslash\lambda y.\mathsf{id}\ y] \\
&\xrightarrow{\mathsf{lsv}}_{\mathsf{sn}} ((\lambda y.\underline{\mathsf{id}\ y})\ w)[w\backslash\lambda y.\mathsf{id}\ y] \\
&\xrightarrow{\mathsf{db}}_{\mathsf{sn}} \underline{((\lambda y.x[x\backslash y])\ w)}[w\backslash\lambda y.\mathsf{id}\ y] \\
&\xrightarrow{\mathsf{db}}_{\mathsf{sn}} \underline{x}[x\backslash y][y\backslash w][w\backslash\lambda y.\mathsf{id}\ y] \\
&\xrightarrow{\mathsf{lsv}\times 3}_{\mathsf{sn}} (\lambda y.\underline{\mathsf{id}\ y})[x\backslash\lambda y.\mathsf{id}\ y][y\backslash\lambda y.\mathsf{id}\ y][w\backslash\lambda y.\mathsf{id}\ y] \\
&\xrightarrow{\mathsf{db}}_{\mathsf{sn}} (\lambda y.x[x\backslash y])[x\backslash\lambda y.\mathsf{id}\ y][y\backslash\lambda y.\mathsf{id}\ y][w\backslash\lambda y.\mathsf{id}\ y]
\end{aligned}$$

Notice how $\mathsf{id}\ y$ 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 Ω a diverging term) to ever be substituted in the

$$\text{VAR}\ \frac{x \in \varphi \cup \omega}{x \in \mathcal{N}_{\varphi,\omega,\mu}} \qquad \text{ABS-}\top\ \frac{t \in \mathcal{N}_{\varphi\cup\{x\},\omega,\top}}{\lambda x.t \in \mathcal{N}_{\varphi,\omega,\top}} \qquad \text{ABS-}\bot\ \frac{t \in \mathcal{N}_{\varphi,\omega\cup\{x\},\bot}}{\lambda x.t \in \mathcal{N}_{\varphi,\omega,\bot}} \qquad \text{ES}\ \frac{t \in \mathcal{N}_{\varphi,\omega,\mu}}{t[x \setminus u] \in \mathcal{N}_{\varphi,\omega,\mu}}$$

$$\text{APP-}\varphi\ \frac{t \in \mathcal{N}_{\varphi,\omega,\mu} \qquad t \in \mathcal{S}_{\varphi} \qquad u \in \mathcal{N}_{\varphi,\omega,\top}}{t\ u \in \mathcal{N}_{\varphi,\omega,\mu}} \qquad \text{APP-}\omega\ \frac{t \in \mathcal{N}_{\varphi,\omega,\mu} \qquad t \in \mathcal{S}_{\omega}}{t\ u \in \mathcal{N}_{\varphi,\omega,\mu}}$$

$$\text{ES-}\varphi\ \frac{t \in \mathcal{N}_{\varphi\cup\{x\},\omega,\mu} \qquad u \in \mathcal{N}_{\varphi,\omega,\bot} \qquad u \in \mathcal{S}_{\varphi}}{t[x \setminus u] \in \mathcal{N}_{\varphi,\omega,\mu}} \qquad \text{ES-}\omega\ \frac{t \in \mathcal{N}_{\varphi,\omega\cup\{x\},\mu} \qquad u \in \mathcal{S}_{\omega}}{t[x \setminus u] \in \mathcal{N}_{\varphi,\omega,\mu}}$$

Figure 11: Local normal forms.

following example, thus preventing normalization (with a a closed term).

$$\begin{array}{rcl} \underline{w}\ (\lambda x.a)[w\backslash\lambda y.y\ \Omega] & \xrightarrow{\mathsf{lsv}}_{\mathsf{sn}} & \underline{(\lambda y.y\ \Omega)\ (\lambda x.a)}[w\backslash\lambda y.y\ \Omega] \\ & \xrightarrow{\mathsf{db}}_{\mathsf{sn}} & (\underline{y}\ \Omega)[y\backslash\lambda x.a][w\backslash\lambda y.y\ \Omega] \\ & \xrightarrow{\mathsf{lsv}}_{\mathsf{sn}} & (\underline{(\lambda x.a)\ \Omega})[y\backslash\lambda x.a][w\backslash\lambda y.y\ \Omega] \\ & \xrightarrow{\mathsf{db}}_{\mathsf{sn}} & a[x\backslash\Omega][y\backslash\lambda x.a][w\backslash\lambda y.y\ \Omega] \end{array}$$

Notice how the sequence of reductions has progressively removed all the occurrences of Ω, 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 Figure 11.

If an abstraction is in a $\top$ position, its variable is added to the set φ of frozen variables, as in Figure 6. But if an abstraction is in a $\bot$ position, its variable is added to a new set ω, as shown in rule ABS-$\bot$ of Figure 11. 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 ω), our λ_{sn}-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 APP-ω of Figure 11. 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 ES-ω.

Example. The justification that our example argument $\lambda y.y\ \Omega$ is a local normal form in a *non*-top-level-like position is summed up by the following derivation.

$$\dfrac{\dfrac{\text{VAR}\ \dfrac{y \in \{y\}}{y \in \mathcal{N}_{\emptyset,\{y\},\bot}} \qquad y \in \mathcal{S}_{\omega}}{y\ \Omega \in \mathcal{N}_{\emptyset,\{y\},\bot}}\ \text{APP-}\omega}{\lambda y.y\ \Omega \in \mathcal{N}_{\emptyset,\emptyset,\bot}}\ \text{ABS-}\bot$$

The definition of local normal forms (Figure 11) is consistent with the already given definition of normal forms (Figure 8). In a sense, normal forms are top-level local normal forms.

Lemma 5.1 (Local normal forms)**.** *$t \in \mathcal{N}_\varphi$ if and only if $t \in \mathcal{N}_{\varphi,\emptyset,\top}$.*

Proof. Each direction is by induction on the derivation of (local) normality. Dealing with the rule ES-φ requires a lemma stating that when $u \in \mathcal{S}_\varphi$, we have $u \in \mathcal{N}_{\varphi,\omega,\top}$ if and only if $u \in \mathcal{N}_{\varphi,\omega,\bot}$. □

This implies that the two calculi λ_{sn} and $\lambda_{\mathsf{sn}+}$ have the same normal forms, and that the completeness theorem stated for λ_{sn} (Theorem 4.10) is still valid for the restricted calculus $\lambda_{\mathsf{sn}+}$.

5.2. **Diamond property.** As mentioned before, in both λ_{c} and λ_{sn}, terms might be substituted as soon as they are values, thus potentially causing duplicate computations, and even breaking confluence. Things are different however in our restricted calculus $\lambda_{\mathsf{sn}+}$.

Consider for instance the counterexample to confluence given at the end of Section 4.1. Given the term $x[x\backslash\lambda y.(\lambda z.z)\ y]$, the calculus λ_{sn} allows both substituting the value $\lambda y.(\lambda z.z)\ y$ or reducing it further. The restricted calculus $\lambda_{\mathsf{sn}+}$, however, prevents the substitution of $\lambda y.(\lambda z.z)\ y$, since this value is not a local normal form. Hence, $\lambda_{\mathsf{sn}+}$ allows only one of the two paths that are possible in λ_{sn}, which is enough to restore confluence. Actually, $\lambda_{\mathsf{sn}+}$ even enjoys the stronger *diamond property*, that is, confluence in one step.

Theorem 5.2 (Diamond)**.** *Suppose $t \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_1$ and $t \xrightarrow{\rho_2,\varphi,\mu}_{\mathsf{sn}+} t_2$ with $t_1 \neq t_2$. Assume that, if ρ_1 and ρ_2 are* sub *or* id*, then they apply to separate variables and that, if ρ_1 or ρ_2 is* sub*, then it applies to a variable that is not in φ. Then there exists t' such that $t_1 \xrightarrow{\rho_2,\varphi,\mu}_{\mathsf{sn}+} t'$ and $t_2 \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t'$.*

Proof. The statement of the diamond theorem has first to be generalized so that the steps $t \to t_1$ and $t \to t_2$ can use the main reduction $\xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}}$ or the auxiliary reductions $\to_{\mathsf{dB}}$ and $\xrightarrow{\varphi,\mu}_{\mathsf{lsv}}$. The proof is then an induction on the size of t, with tedious reasoning by case on the shape of t and on the last inference rule applied on each side. Most cases are rather unsurprising. We present here selected subcases applying to the shape $t[x\backslash u]$, which illustrate the main subtleties encountered in the whole proof.

- Case ES-LEFT vs ES-RIGHT. More precisely we have
  - $t[x\backslash u] \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_1[x\backslash u]$ by rule ES-LEFT with $t \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_1$, and
  - $t[x\backslash u] \xrightarrow{\rho_2,\varphi,\mu}_{\mathsf{sn}+} t[x\backslash u_2]$ by rule ES-RIGHT with $t \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}+} t$ and $u \xrightarrow{\rho_2,\varphi,\mu}_{\mathsf{sn}+} u_2$.

  Notice that, by freshness of x, the rule ρ_1 cannot be a sub or id applying to the variable x. Then we can apply our induction hypothesis to $t \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_1$ and $t \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}+} t$ to deduce that $t_1 \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}+} t_1$. With rule ES-RIGHT we then deduce $t_1[x\backslash u] \xrightarrow{\rho_2,\varphi,\mu}_{\mathsf{sn}+} t_1[x\backslash u_2]$. Moreover, the rule ES-LEFT applied to $t \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_1$ immediately gives $t[x\backslash u_2] \xrightarrow{\rho_2,\varphi,\mu}_{\mathsf{sn}+} t_1[x\backslash u_2]$ and closes the case.
- Case ES-LEFT-$\mathcal{S}$ vs ES-RIGHT. This is superficially similar to the previous one, but adds some subtleties, as we now have
  - $t[x\backslash u] \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_1[x\backslash u]$ by rule ES-LEFT-$\mathcal{S}$ with $u \in \mathcal{S}_\varphi$ and $t \xrightarrow{\rho_1,\varphi\cup\{x\},\mu}_{\mathsf{sn}+} t_1$, and
  - $t[x\backslash u] \xrightarrow{\rho_2,\varphi,\mu}_{\mathsf{sn}+} t[x\backslash u_2]$ by rule ES-RIGHT with $t \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}+} t$ and $u \xrightarrow{\rho_2,\varphi,\mu}_{\mathsf{sn}+} u_2$.

The reduction $t \xrightarrow{\rho_1,\varphi\cup\{x\},\mu}_{\mathsf{sn}+} t_1$ uses the frozen variables $\varphi \cup \{x\}$ rather than just φ. To apply the induction hypothesis, we first have to weaken $t \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}+} t$ into $t \xrightarrow{\mathsf{id}_x,\varphi\cup\{x\},\mu}_{\mathsf{sn}+} t$ (Lemma 5.4 below). The induction hypothesis then gives $t_1 \xrightarrow{\mathsf{id}_x,\varphi\cup\{x\},\mu}_{\mathsf{sn}+} t_1$, which then has to be strengthened into $t_1 \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}+} t_1$ (Lemma 5.5 below) to obtain $t_1[x\backslash u] \xrightarrow{\rho_2,\varphi,\mu}_{\mathsf{sn}+} t_1[x\backslash u_2]$ with rule ES-RIGHT.

Finally, to close the case with rule ES-LEFT-$\mathcal{S}$ we also need to justify that $u_2 \in \mathcal{S}_\varphi$, which we do using a stability property of structures (Lemma 5.3 below).

- Case ES-LEFT-$\mathcal{S}$ vs LSV. These cases are not compatible. Indeed ES-LEFT-$\mathcal{S}$ applies to a term $t[x\backslash u]$ where $u \in \mathcal{S}_\varphi$, whereas LSV and the associated auxiliary rules apply to a term $t[x\backslash u]$ where u is an answer.
- Case ES-LEFT vs LSV-BASE. In this case our starting term has necessarily the shape $t[x\backslash v]$ with v a value and we have
  – $t[x\backslash v] \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_1[x\backslash v]$ by rule ES-LEFT with $t \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_1$, and
  – $t[x\backslash v] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t_2[x\backslash v]$ by rule LSV-BASE with $t \xrightarrow{\mathsf{sub}_{x\backslash v},\varphi,\mu}_{\mathsf{sn}} t_2$.
  Due to freshness, the variable x cannot appear in ρ_1 nor in φ. Then by induction hypothesis we obtain t_3 such that $t_1 \xrightarrow{\mathsf{sub}_{x\backslash v},\varphi,\mu}_{\mathsf{sn}} t_3$ and $t_2 \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}} t_3$. By applying rule LSV-BASE to the first reduction we obtain $t_1[x\backslash v] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t_3[x\backslash v]$, and by applying rule ES-LEFT to the second reduction we obtain $t_2[x\backslash v] \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}} t_3[x\backslash v]$ and conclude.
- Case ES-LEFT vs LSV-σ. In this case our starting term has necessarily the shape $t[x\backslash u[y\backslash w]]$ and we have
  – $t[x\backslash u[y\backslash w]] \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_1[x\backslash u[y\backslash w]]$ by rule ES-LEFT with $t \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_1$, and
  – $t[x\backslash u[y\backslash w]] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t_2[y\backslash w]$ by rule LSV-σ with $t[x\backslash u] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t_2$.
  From the premise $t \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_1$, by rule ES-LEFT we deduce $t[x\backslash u] \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_1[x\backslash u]$. Since the term $t[x\backslash u]$ is smaller than $t[x\backslash u[y\backslash w]]$, we apply the induction hypothesis to obtain a term t_3 such that $t_1[x\backslash u] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t_3$ and $t_2 \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_3$.

  From the first reduction, by rule LSV-σ we deduce $t_1[x\backslash u[y\backslash w]] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t_3[y\backslash w]$ and from the second reduction, by rule ES-LEFT we deduce $t_2[y\backslash w] \xrightarrow{\rho_1,\varphi,\mu}_{\mathsf{sn}+} t_3[y\backslash w]$. These two final reductions close the diamond on the term $t_3[y\backslash w]$. □

The following facts were used in the proof of Theorem 5.2, but they have merits on their own, especially Lemma 5.3 of stability of structures by reduction.

Lemma 5.3 (Stability of structures)**.** *Suppose $t \in \mathcal{S}_\varphi$ and $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}+} t'$. Assume that, if ρ is sub, it applies to a variable that is not in φ. Then $t' \in \mathcal{S}_\varphi$.*

Proof. By induction on $t \in \mathcal{S}_\varphi$, with cases on the last inference rule of the derivation of $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}+} t'$. □

The restriction when ρ is of kind sub has one simple (but crucial) goal: ruling out a direct substitution of the leading variable of the structure t, such as $x\ u \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}+} (\lambda y.y)\ u$ with $\rho = \mathsf{sub}_{x\backslash\lambda y.y}$ and $\varphi = \{x\}$. Indeed, any structure must be led by a *frozen* variable, that is a variable of which we know that it cannot be substituted. It can be checked that no valid lsv reduction can be related to this forbidden case of sub.

Lemma 5.4 (Weakening)**.** *Suppose $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}+} t'$ and $\varphi \subseteq \varphi'$. Assume that, if ρ is sub, it applies to a variable that is not in φ'. Then $t \xrightarrow{\rho,\varphi',\mu}_{\mathsf{sn}+} t'$.*

Proof. By induction on the derivation of $t \xrightarrow{\rho,\varphi',\mu}_{\mathsf{sn}+} t'$. □

Lemma 5.5 (Strengthening)**.** *If* $t \xrightarrow{\mathsf{id}_x,\varphi\cup\{x\},\mu}_{\mathsf{sn}+} t'$*, then* $t \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}+} t'$*.*

Proof. By induction on the derivation of $t \xrightarrow{\rho,\varphi',\mu}_{\mathsf{sn}+} t'$. There are two subtle cases with the rules APP-RIGHT and ES-LEFT-$\mathcal{S}$, which both rely on the following property: If $t \in \mathcal{S}_{\varphi\cup\{x\}}$ and $t \notin \mathcal{S}_\varphi$, then $t \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}+} t'$. This property is proved by induction on $t \in \mathcal{S}_{\varphi\cup\{x\}}$. □

5.3. **Relative optimality.** The $\lambda_{\mathsf{sn}+}$-calculus is a restriction of λ_{sn} that requires terms to be eagerly reduced to local normal form before they can be substituted (Figure 10). This eager reduction is never wasted. Indeed, λ_{sn} (and *a fortiori* its subset $\lambda_{\mathsf{sn}+}$) only reduces needed redexes, that is, redexes that are necessarily reduced in any reduction to normal form. As a consequence, reductions in $\lambda_{\mathsf{sn}+}$ are never longer than equivalent reductions in λ_{sn}. On the contrary, by forcing some reductions to be performed before a term is substituted (*i.e.*, potentially duplicated), this strategy produces in many cases reduction sequences that are strictly shorter than the ones given by the original strong call-by-need strategy [BBBK17].

Theorem 5.6 (Minimality)**.** *With* $t' \in \mathcal{N}_\varphi$*, if* $t \to^n_{\mathsf{sn}} t'$ *and* $t \to^m_{\mathsf{sn}+} t'$ *then* $m \leqslant n$*.*

Remark that this minimality result is relative to λ_{sn}. The reduction sequences of $\lambda_{\mathsf{sn}+}$ are not necessarily optimal with respect to the unconstrained λ_{c} or λ-calculi. For instance, neither $\lambda_{\mathsf{sn}+}$ nor λ_{sn} allow reducing r in the term $(\lambda x.x\ (x\ a))\ (\lambda y.y\ r)$ prior to its duplication.

6. Formalization in Abella

We used the Coq proof assistant for our first attempts to formalize our results. We experimented both with the locally nameless approach [Cha12] and parametric higher-order abstract syntax [Chl08]. 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 [BCG+14], in the hope that it would make syntactic proofs more straightforward. This section describes our experience with this tool.

6.1. **Nominal variables and λ-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 a normal form of λ_{sn} and $\lambda_{\mathsf{sn}+}$, as given in Figure 8. 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.

```
Define nf : trm -> prop by
  nabla x, nf x;
  nf (abs U) := nabla x, nf (U x);
  ...
```

Note that Abella is based on a λ-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`.

```
forall T U, nabla x,
struct T x -> star T U -> 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, the design of Abella makes it much easier to use theorem statements in which universal quantifiers happen before nominal ones, 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:

```
forall T U, nabla x,
struct (T x) x -> star (T x) (U x) -> 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 λ-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 by the following statement:

```
star (T x) (U x) -> exists V, U = (y\ V).
```

The λ-tree `y\V` can be understood as the anonymous function $y \mapsto V$. Thus, the equality above forces `U` to be a constant λ-tree, since y does not occur in V, by lexical scoping.

6.2. **Functions and relations.** Our Abella formalization assumes a type `trm` and three predefined ways to build elements of that type: application, abstraction, and explicit substitution.

```
type app trm -> trm -> trm.
type abs (trm -> trm) -> trm.
type es  (trm -> trm) -> trm -> trm.
```

For example, a term $t[x\backslash u]$ of our calculus will be denoted `(es (x\t) u)` with `t` containing some occurrences of `x`. Again, `x\t` is Abella's notation for a λ-tree `t` in which `x` has to be substituted (by application). It can thus be understood as an anonymous function $x \mapsto t$ in the meta-system.

Since Abella does not provide proper functions, we instead use a relation `star` to define the unfolding function from λ_{c} to λ. We define it in the specification logic using λ-Prolog rules (`pi` is the universal quantifier in the specification logic).

```
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).
```

Of particular interest is the way binders are handled; they are characterized by stating that they are their own image: `star x x`.

Since `star` 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 λ-terms, and that it maps to exactly one λ-term. Needless to say, all of that would be simpler if Abella had native support for functions.

6.3. **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.

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

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 |- nf (abs T)`}. For simplicity, let us assume that the last three rules of `nf` do not exist (that is, only `X` and `abs U` are covered). In that case, there is only three ways of deriving the proposition. Indeed, it can be derived from {`L |- frozen (abs T)`} (first rule). It can also be derived from `nabla x,` {`L, frozen x |- nf (T x)`} (second rule). Finally, the third way to derive it is if `nf (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.*, some term `T` whose bound variable was replaced with `U` satisfies predicate `P` in context `L`. The simplest way is if one can prove `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 (*e.g.*, `struct T x`) but with respect to all the frozen variables (*e.g.*, {`frozen x |- 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 Figure 5 for example.

Performing specification-level proofs does not come without its own set of issues, though. As explained earlier, a proposition {`L |- nf (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, *e.g.*, `L` 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.

6.4. **Formal definitions.** Let us now describe the main definitions of our Abella formalization. In addition to `nf` and `star` which have already been presented, we have a relation `step` which formalizes the reduction rules of λ_{sn} and $\lambda_{\mathsf{sn}+}$ presented in Figure 6.

```
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 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 respect to Section 4 is the predicate `active`, which characterizes the variable being considered by id_x (`idx`) and $\mathsf{sub}_{x \setminus v}$ (`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 ABS-$\bot$. While the antecedent of the rule is at position $\bot$, 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 $\lambda_{\mathsf{sn}+}$, as given in Figures 7 and 10 for rule LSV-BASE, are as follows.

```
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:

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

These various rules make use of structures (`struct`), as given in Figure 5.

```
struct X :- frozen X.
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 Figure 11 are as follows. As for the `step` relation, one of the rules for abstraction was generalized with respect to Section 5. 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\ 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 ω use a dedicated predicate `struct_omega`, which is just a duplicate of `struct`. Another approach, perhaps more elegant, would have been to parameterize `struct` with either `frozen` or `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.
```

Notice that our Abella definition of reduction is more permissive than the restricted calculus $\lambda_{\mathsf{sn}+}$, as it allows the set ω to be non-empty when the check `lnf bot (abs V)` is performed in the definition of `aux_lsv`, whereas the corresponding rule LSV-BASE in the calculus requires an empty set.

The main reason for this discrepancy is that all the variables of a term need to appear one way or another in the context. So, we have repurposed `omega` for variables that are neither `active` nor `frozen`. This has avoided the introduction of yet another predicate in the formalization. But this comes at the expense of the set ω being potentially non-empty when evaluating whether a term is a local normal form.

As a consequence, more terms will pass the test and be substituted. However, this version is still included in the λ_{sn}-calculus, and thus fully supports the claims of Section 4.2. If one were to consider a formalization of completeness instead, this discrepancy would get in the way.

Extra definitions. Having a characterization of λ_{sn}-terms is sometimes useful, as it allows 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.
```

Similarly, we might need to characterize pure λ-terms.

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

Finally, let us remind the definitions of a β-reduction, of a sequence `betas` of zero or more β-reductions, and of the normal forms `nfb` of the λ-calculus, as they will be needed to state the main theorems.

```
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).

betas M M.
betas M N :- beta M P, betas P N.

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.
notabs (app T U).
```

6.5. **Formally verified properties.** To conclude this section on our Abella formalization, let us state the theorems that were fully proved using Abella. First comes the simulation property (Lemma 4.1), which states that, if $T \to_{\mathsf{sn}+} U$, then $T^\star \to_\beta^* U^\star$.

```
Theorem simulation' : forall T U T* U*,
  {star T T*} -> {star U U*} -> {red T U} -> {betas T* U*}.
```

Then comes the fact that (local) normal forms are exactly the terms that are not reducible in $\lambda_{\mathsf{sn}+}$ (Lemma 4.2).

```
Theorem lnf_nand_red : forall T U,
  {lnf top T} -> {red T U} -> false.
Theorem nf_nand_red : forall T U,
  {nf T} -> {red T U} -> false.

Theorem lnf_or_red : forall T,
  {trm T} -> {lnf top T} \/ exists U, {red T U}.
Theorem nf_or_red : forall T,
  {trm T} -> {nf T} \/ exists U, {red T U}.
```

Finally, if T is a normal form of λ_{sn}, then $T^\star$ is a normal form of the λ-calculus (Lemma 4.3).

```
Theorem nf_star' : forall T T*,
  {nf T} -> {star T T*} -> {nfb T*}.
```

7. Abstract machine

We have also implemented an abstract machine for $\lambda_{\mathsf{sn}+}$, so as to get more insights about its reduction rules. Indeed, while Figures 6 and 7 already give a presentation of the calculus that is close to be implementable, a number of questions remain. In particular, several rules have side conditions. So, even if the calculus is relatively optimal in the number of reductions, it does not tell much about its practical efficiency. For example, rules APP-RIGHT and ES-LEFT-$\mathcal{S}$ require subterms to be structures, while rule ES-RIGHT involves the reduction id_x, which does not progress by definition. As for rule LSV-BASE of Figure 10, it requires a subterm to be both a value and in local normal form. Thus, an actual implementation might be nowhere near the ideal time complexity of $O(n + m)$, with n the size of the input term and m the number of steps to reach its normal form.

Another practical issue that is not apparent in Figures 6 and 7 is the handling of explicit substitutions. Indeed, the ability to freely reduce under abstractions makes them especially brittle, as the normalization status of their right-hand side keep switching back and forth as variables become frozen. This phenomenon does not occur when performing a weak reduction, even when iterating it to get the strong normal form.

Finally, actually implementing the calculus might make apparent some reductions in Figures 6 and 7 that needlessly break sharing between subterms.

Configurations of the machine. Whether a big-step or small-step semantics is chosen to present the abstract machine has no impact on its effectiveness. So, for readability, we describe its big-step semantics (Figure 12), assuming the term being reduced has a normal form. Since the reduction rules do not perform any fancy trick, it could be turned into a small-step semantics using an explicit continuation, in a traditional fashion.

Configurations (Σ, Π, μ, t) are formed from an environment Σ of variables, a stack Π of terms, a mode μ, and the currently visited term t. Assuming the machine terminates, it returns a pair (Σ', t'), comprised of a new environment Σ' as well as a term t' equivalent to t, hopefully reduced. Notice that all the rules of Figure 12 propagate the environment Σ along the computations and never duplicate it in any way, so it should be understood as a global mutable store.

The argument stack Π is used for applications; it is initially empty. When the abstract machine needs to reduce an application $t\ u$, it pushes u on the stack Π, and then focuses on reducing t (rule APP). This stack appears on the left of the rules, but never on the right, as it is always fully consumed by the machine. Contrarily to the variable environment Σ which is global, new stacks might be created on the fly to reduce some subterms. This is not a strong requirement, though, as an implementation could also use a global stack, by keeping track of the current number of arguments with a counter or a stack mark.

The environment Σ associates some data to every free variable x of the currently visited term t. If x was originally bound by an explicit substitution, *e.g.*, $t[x\backslash u]$, then the environment has been extended with a new binding, which is denoted by $\Sigma; x \mapsto u$ in rule ES. The term associated to variable x in the environment Σ is denoted $\Sigma(x)$, as in rule VAR-ABS. This association can evolve along the computations, as the associated term u might be replaced by some reduced term u'. In that case, the resulting environment is denoted by $\Sigma[x \mapsto u']$ as illustrated by rule VAR-$\mathcal{S}$. This association is eventually removed from the environment, as shown by rule ES.

If x, however, was originally the variable bound by an abstraction, *e.g.*, $\lambda x.t$, then the environment associates to it a special symbol to denote whether x is frozen ($\lambda_\top$) or not ($\lambda_\bot$), as illustrated by rule ABS-NIL. Initially, the environment associates $\lambda_\top$ to every free variable of the term, if any.

Finally, the mode μ of a configuration is reminiscent of the mode $\top$ and $\bot$ of the calculus. There is a slight difference, though. The currently visited term is in a top-level-like position only if μ is $\top$ and the stack Π is empty. Initially, the mode μ is set to $\top$.

Reduction rules. The machine implements a leftmost outermost strategy. Indeed, the argument u of an application $t\ u$ is moved to the stack while the machine proceeds to the head t (rule APP). Similarly, when an explicit substitution $t[x\backslash u]$ is encountered, the binding $x \mapsto u$ is added to the environment Σ while the focus moves to t (rule ES). Finally, when encountering an abstraction $\lambda x.t$ while the argument stack is non-empty, the top u of the stack is popped to create an explicit substitution $t[x\backslash u]$ (rule ABS-CONS). Note that restricting the machine to a leftmost outermost evaluation strategy is not an issue, thanks to the diamond property.

While rules APP and ABS-CONS could be found in any other abstract machine, rule ES is a bit peculiar. Indeed, once t has been reduced to t', the abstract machine recreates an explicit substitution $t'[x\backslash u']$ with u' the term associated to x (which might be u or some reduced form of it). In other words, the binding is removed from the environment and it might be added back to it again later. This seemingly wasteful operation would not be needed in a

$$\textsc{abs-cons}\ \frac{(\Sigma, \Pi, \mu, t[x\backslash u]) \to (\Sigma', t')}{(\Sigma, u \cdot \Pi, \mu, \lambda x.t) \to (\Sigma', t')} \qquad \textsc{abs-nil}\ \frac{(\Sigma; x \mapsto \lambda_\mu, \mathit{Nil}, \mu, t) \to (\Sigma'; x \mapsto \lambda_\mu, t')}{(\Sigma, \mathit{Nil}, \mu, \lambda x.t) \to (\Sigma', \lambda x.t')}$$

$$\textsc{app}\ \frac{(\Sigma, u \cdot \Pi, \mu, t) \to (\Sigma', t')}{(\Sigma, \Pi, \mu, t\ u) \to (\Sigma', t')} \qquad \textsc{es}\ \frac{(\Sigma; x \mapsto u, \Pi, \mu, t) \to (\Sigma'; x \mapsto u', t')}{(\Sigma, \Pi, \mu, t[x\backslash u]) \to (\Sigma', t'[x\backslash u'])}$$

$$\textsc{var-}\lambda\textsc{-Frozen}\ \frac{\Sigma(x) = \lambda_\top \qquad (\Sigma, \Pi, \mu, x) \to_{\mathcal{S}_\varphi} (\Sigma', t')}{(\Sigma, \Pi, \mu, x) \to (\Sigma', t')} \qquad \textsc{var-}\lambda\textsc{-NonFrozen}\ \frac{\Sigma(x) = \lambda_\bot \qquad (\Sigma, \Pi, \mu, x) \to_{\mathcal{S}_\omega} (\Sigma', t')}{(\Sigma, \Pi, \mu, x) \to (\Sigma', t')}$$

$$\textsc{var-abs}\ \frac{\Sigma(x) = u \qquad (\Sigma, \mathit{Nil}, \bot, u) \to (\Sigma', u') \qquad u' = (\lambda y.t)\mathcal{L} \qquad (\Sigma'[x \mapsto u'], \Pi, \mu, u') \to (\Sigma'', u'')}{(\Sigma, \Pi, \mu, x) \to (\Sigma'', u'')}$$

$$\textsc{var-}\mathcal{S}\ \frac{\Sigma(x) = u \qquad (\Sigma, \mathit{Nil}, \bot, u) \to (\Sigma', u') \qquad \Sigma' \vdash u' \in \mathcal{S}_\alpha \qquad (\Sigma'[x \mapsto u'], \Pi, \mu, x) \to_{\mathcal{S}_\alpha} (\Sigma'', u'')}{(\Sigma, \Pi, \mu, x) \to (\Sigma'', u'')}$$

$$\textsc{s-phi}\ \frac{(\Sigma_0, \mathit{Nil}, \top, t_1) \to (\Sigma_1, t_1') \qquad \ldots \qquad (\Sigma_{n-1}, \mathit{Nil}, \top, t_n) \to (\Sigma_n, t_n')}{(\Sigma_0, t_1 \cdot \ldots \cdot t_n, \mu, x) \to_{\mathcal{S}_\varphi} (\Sigma_n, x\ t_1' \ldots t_n')}$$

$$\textsc{s-omega}\ \frac{}{(\Sigma, t_1 \cdot \ldots \cdot t_n, \mu, x) \to_{\mathcal{S}_\omega} (\Sigma, x\ t_1 \ldots t_n)}$$

Figure 12: Big-step rules for the abstract machine.

$$\frac{}{\Sigma; x \mapsto \lambda_\top \vdash x \in \mathcal{S}_\varphi} \qquad \frac{}{\Sigma; x \mapsto \lambda_\bot \vdash x \in \mathcal{S}_\omega}$$

$$\frac{\Sigma \vdash t \in \mathcal{S}_\alpha}{\Sigma; x \mapsto t \vdash x \in \mathcal{S}_\alpha} \qquad \frac{\Sigma \vdash t_1 \in \mathcal{S}_\alpha}{\Sigma \vdash t_1\ t_2 \in \mathcal{S}_\alpha} \qquad \frac{\Sigma; x \mapsto u \vdash t \in \mathcal{S}_\alpha}{\Sigma \vdash t[x\backslash u] \in \mathcal{S}_\alpha}$$

Figure 13: Structures, as viewed by the abstract machine.

weak call-by-need strategy, but here it is critical. Indeed, in strong call-by-need, the term u' might refer to some abstraction variable y. If this abstraction is later applied multiple times, u' would further be reduced and it would mix up the multiple values associated to y. Here is an example:

$$(\lambda f.c_1\ (f\ c_2)(f\ c_3))\ (\lambda y.(c_4\ x)[x\backslash y])$$

When reducing an abstraction $\lambda x.t$ while the stack Π is empty, the binding $x \mapsto \lambda_\mu$ is added to the environment and the machine proceeds to reducing t (rule ABS-NIL). This is the only rule where the mode μ really matters, as it is used to guess whether a variable

is frozen. Once t has been reduced to t', the final value is obtained by reconstructing the abstraction $\lambda x.t'$. In an abstract machine using weak reduction, a similar rule would exist, except that it would always use $x \mapsto \lambda_\top$, as x would always behave like a free variable.

The more complicated rules are triggered when reducing a variable x. The behavior then depends on the corresponding binding in the environment. If x is a frozen abstraction variable, then the machine creates a structure with x as the head variable (rule VAR-λ-FROZEN). To do so, it reduces all the terms $t_1, \ldots, t_n$ of the argument stack and applies x to them (auxiliary rule S-PHI). If x is an abstraction variable but not (yet) frozen, the terms of the argument stack are still given as arguments to x, but without being first reduced (rule VAR-λ-NONFROZEN and auxiliary rule S-OMEGA).

The last case corresponds to a variable x bound to a term u. The first step is to reduce u to u' and to update the binding of x (rules VAR-ABS and VAR-$\mathcal{S}$). Depending on the form of u', the evaluation will then proceed in different ways. If u' is an abstraction (possibly hidden behind layers of explicit substitutions $\mathcal{L}$), then the evaluation proceeds with it, so as to consume the argument stack if not empty (rule VAR-ABS). If u' is a structure (as characterized by the rules of Figure 13), then two subcases are considered, depending on whether u' is a structure with respect to the set φ or to the set ω. In the former case, all the terms of the argument stack are reduced before being aggregated to x (rule VAR-$\mathcal{S}$ and auxiliary rule S-PHI). In the latter case, the terms of the argument stack are aggregated to x without being first reduced (rule VAR-$\mathcal{S}$ and auxiliary rule S-OMEGA).

Validation. The rules of our calculus can be applied to arbitrary subterms of the λ-term being reduced. But when it comes to the abstract machine, the reduction rules are only applied to the currently visited subterm. Thus, even if the calculus rules are correctly implemented in the machine, if some subterms were never visited or at the wrong time, the final term might not be reduced as much as possible. This is illustrated by the auxiliary rules S-PHI and S-OMEGA. The latter rule can get away with not reducing the application arguments, as this term will necessarily be visited again at a later time, once the status of x becomes clear. But for rule S-PHI, this might be the last time the term is ever visited, so it has to be put into normal form.

The correctness of the abstract machine would thus warrant another Abella formalization. Short of such a formal verification, another way to increase the confidence in this abstract machine is to validate it. We did so by applying it to all the λ-terms of depth 5. The normal forms we obtained were compared to the ones from a machine implementing a straightforward call-by-name strategy. If none of the machines finished after 1500 reduction steps, we assumed that the original term had no normal form and that both machines agreed on it. No discrepancies were found for these millions of λ-terms, hence providing us with a high confidence in the correctness of our abstract machine.

Implementation details. In our implementation, variables are represented by their De Bruijn indices. Since there is a strict push/pop discipline for the environment, as illustrated by rules ABS-NIL and ES, Σ has been implemented using a random-access list, and the De Bruijn indices point into it. The target cells contain either $\lambda_\top$, $\lambda_\bot$, or the term associated to the given variable. This term is stored in a mutable cell, so that any reduction to it are retained for later use.

Using De Bruijn indices makes it trivial to access the environment, but it also means that they have to be kept consistent with the current environment. Indeed, the environment

might have grown since the time an application argument was put into the stack by rule APP, and similarly with the bindings of the environment itself. Thus, the stack and the environment actually contain *closures*, *i.e.*, pairs of a term u and the size of the environment at the time u was put into the stack or environment. Then, a rule such as ABS-CONS can adjust the De Bruijn indices of u to the current environment. In a traditional way, our implementation performs this adjustment lazily, that is, a mark is put on u and the actual adjustment is only performed once u is effectively traversed. So, when going from $\lambda x.t$ to $t[x\backslash u]$ in rule ABS-CONS, the subterm t is left completely unchanged, while the subterm u is just marked as needing an adjustment of its De Bruijn indices.

Since rules VAR-ABS and VAR-$\mathcal{S}$ systematically force a reduction of the term u bound to x whenever x is encountered, there is a glaring inefficiency. Indeed, any evolution of the environment between two occurrences of x only concerns variables that are meaningless to x. So, while the first encounter of x puts u into local normal form, any subsequent encounter will traverse u without modifying it any further. As a consequence, the implementation differs from Figure 12 in that the environment also stores the status of u. Initially, u has status *non-evaluated*. After its first reduction, the status of u changes to one of the following three *evaluated* statuses: an abstraction $(\lambda y.t)\mathcal{L}$, a structure of $\mathcal{S}_\omega$, a structure of $\mathcal{S}_\varphi$. That way, the implementation of VAR-ABS and VAR-$\mathcal{S}$ can directly skip to executing the rightmost antecedent, depending on the status. Moreover, the status stored in the environment is not obtained by a traversal of u; this information is directly obtained from the previous evaluation. For example, rule ABS-NIL necessarily returns an abstraction, while rule ES returns whatever it obtained from the execution of its antecedent. Thus, our implementation does not needlessly traverse terms.

Observations. First of all, the abstract machine is unfortunately not a faithful counterpart to our calculus. The discrepancy lies in rule VAR-ABS. Indeed, contrarily to rule LSV-BASE of Figure 10, the substituted term is not just a value, it is potentially covered by explicit substitutions: $u' = (\lambda y.t)\mathcal{L}$. As a consequence, when it comes to the explicit substitutions of $\mathcal{L}$, sharing between both occurrences of u' in the configuration $(\Sigma'; x \mapsto u', \Pi, \mu, u')$ is lost. It is not clear yet what the cost of recovering this sharing is.

Rules VAR-ABS and VAR-$\mathcal{S}$ of the abstract machine also put into light some inefficiencies of the corresponding rules of our calculus. Assume that the machine encounters x in mode $\top$, that the stack Π is empty, and that x is bound to u in the environment Σ. The machine first evaluates u with mode $\bot$ and updates Σ with the result u'. It then proceeds to evaluating u' in place of x with mode $\top$. But at that point, the binding of x in the environment is no longer updated; it still points to u', not to u''. Thus, the machine will have to reduce u' into u'' for other occurrences of x at top-level-like positions. It would have been more efficient to originally evaluate u in mode $\top$, contrarily to what rule ES-RIGHT of Figure 6 mandates.

Once this inefficiency is solved, another one becomes apparent, again with rule VAR-ABS. The abstract machine evaluates u' (which is then an abstraction) in place of x. But there is not much point in doing so, if the argument stack is empty. It could have instead returned x, since the binding $x \mapsto u'$ is part of the environment, thus preserving some more sharing.

Fixing these two inefficiencies in the abstract machine means that it would implement the reduction rules of a different calculus. This new calculus would not be much different from the one presented in this paper, but it would no longer have the diamond property.

8. Conclusion

This paper presents a λ-calculus dedicated to strong reduction. In the spirit of a call-by-need strategy with explicit substitutions, it builds on a linear substitution calculus [ABKL14]. 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 [BBBK17]. 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 [BBBK17]. And more generally, this new presentation of call-by-need reduction could be useful even in the traditional weak setting.

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, as made apparent by the rules of the abstract machine, our calculus could avoid substituting variables that are not applied (rule lsv-base), following [Yos93, ACSC21], 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 [Bal12].

References

[ABKL14] Beniamino Accattoli, Eduardo Bonelli, Delia Kesner, and Carlos Lombardi. A nonstandard standardization theorem. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, page 659–670, 2014. doi:10.1145/2535838.2535886.

[ABM14] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. Distilling abstract machines. In *19th ACM SIGPLAN International Conference on Functional Programming*, page 363–376, 2014. doi:10.1145/2628136.2628154.

[ABM15] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. A strong distillery. In Xinyu Feng and Sungwoo Park, editors, *Programming Languages and Systems*, volume 9458 of *Lecture Notes in Computer Science*, pages 231–250, 2015. doi:10.1007/978-3-319-26529-2_13.

[ACSC21] Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. Strong call-by-value is reasonable, implosively, February 2021. arXiv:2102.06928.

[AK10] Beniamino Accattoli and Delia Kesner. The structural λ-calculus. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic*, pages 381–395, 2010. doi:10.5555/1887459.1887491.

[AMO+95] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246, 1995. doi:10.1145/199448.199507.

[Bal12] Thibaut Balabonski. A unified approach to fully lazy sharing. In John Field and Michael Hicks, editors, *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 469–480, January 2012. doi:10.1145/2103656.2103713.

[Bal13] Thibaut Balabonski. Weak optimality, and the meaning of sharing. In Greg Morrisett and Tarmo Uustalu, editors, *18th ACM SIGPLAN International Conference on Functional Programming*, pages 263–274, September 2013. doi:10.1145/2500365.2500606.

[BBBK17] Thibaut Balabonski, Pablo Barenbaum, Eduardo Bonelli, and Delia Kesner. Foundations of strong call by need. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi:10.1145/3110264.

[BBCD20] Malgorzata Biernacka, Dariusz Biernacki, Witold Charatonik, and Tomasz Drab. An abstract machine for strong call by value. In Bruno C. d. S. Oliveira, editor, *18th Asian Symposium on Programming Languages and Systems*, volume 12470 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 2020. doi:10.1007/978-3-030-64437-6_8.

[BC19] Malgorzata Biernacka and Witold Charatonik. Deriving an abstract machine for strong call by need. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction*, volume 131 of *Leibniz International Proceedings in Informatics*, pages 8:1–8:20, 2019. doi:10.4230/LIPIcs.FSCD.2019.8.

[BCG+14] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, December 2014. doi:10.6092/issn.1972-5787/4650.

[BKV17] Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Logic Journal of the IGPL*, 25(4):431–464, July 2017. doi:10.1093/jigpal/jzx018.

[BLM21] Thibaut Balabonski, Antoine Lanco, and Guillaume Melquiond. A strong call-by-need calculus. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction*, volume 195 of *Leibniz International Proceedings in Informatics*, pages 9:1–9:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, July 2021. doi:10.4230/LIPIcs.FSCD.2021.9.

[CDC80] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980. doi:10.1305/ndjfl/1093883253.

[CF12] Stephen Chang and Matthias Felleisen. The call-by-need lambda calculus, revisited. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 128–147. Springer Berlin Heidelberg, 2012.

[Cha12] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, October 2012. doi:10.1007/s10817-011-9225-2.

[Chl08] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *13th ACM SIGPLAN International Conference on Functional Programming*, pages 143–156, September 2008. doi:10.1145/1411204.1411226.

[Cré90] Pierre Crégut. An abstract machine for lambda-terms normalization. In *ACM Conference on LISP and Functional Programming*, page 333–340, 1990. doi:10.1145/91556.91681.

[dC07] Daniel de Carvalho. *Sémantiques de la logique linéaire et temps de calcul*. PhD thesis, Université Aix-Marseille II, 2007.

[Gar94] Philippa Gardner. Discovering needed reductions using type theory. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, pages 555–574, 1994.

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *7th ACM SIGPLAN International Conference on Functional Programming*, page 235–246, 2002. doi:10.1145/581478.581501.

[HG91] Carsten K. Holst and Darsten K. Gomard. Partial evaluation is fuller laziness. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, page 223–233, 1991. doi:10.1145/115865.115890.

[Kes09] Delia Kesner. A theory of explicit substitutions with safe and full composition. *Logical Methods in Computer Science*, 5(3), May 2009. doi:10.2168/LMCS-5(3:1)2009.

[Kes16] Delia Kesner. Reasoning about call-by-need by means of types. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 424–441, 2016. doi:10.1007/978-3-662-49630-5_25.

[KV14] Delia Kesner and Daniel Ventura. Quantitative types for the linear substitution calculus. In Josep Diaz, Ivan Lanese, and Davide Sangiorgi, editors, *Theoretical Computer Science*, volume 8705 of *Lecture Notes in Computer Science*, pages 296–310, 2014. doi:10.1007/978-3-662-44602-7_23.

[Mil07] Robin Milner. Local bigraphs and confluence: Two conjectures. *Electronic Notes on Theoretical Computer Science*, 175(3):65–73, June 2007. doi:10.1016/j.entcs.2006.07.035.

[MOW98] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, May 1998. doi:10.1017/S0956796898003037.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

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

[Yos93] Nobuko Yoshida. Optimal reduction in weak-lambda-calculus with shared environments. In *Conference on Functional Programming Languages and Computer Architecture*, page 243–252, 1993. doi:10.1145/165180.165217.