A strong call-by-need calculus

Thibaut Balabonski, Antoine Lanco, Guillaume Melquiond

A strong call-by-need calculus

Thibaut Balabonski
Université Paris-Saclay, CNRS, LMF, France

Antoine Lanco
Université Paris-Saclay, CNRS, Inria, LMF, France

Guillaume Melquiond
Université Paris-Saclay, CNRS, Inria, LMF, France

Abstract

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

The calculus is shown to be *normalizing* in a strong sense: Whenever a λ-term t admits a normal form n in the λ-calculus, then *any* reduction sequence from t in the calculus eventually reaches 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.

2012 ACM Subject Classification Theory of computation → Program semantics; Theory of computation → Operational semantics

Keywords and phrases strong reduction, call-by-need, evaluation strategy, normalization

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Lambda-calculus is seen as the standard model of computation in functional programming languages, once equipped with an *evaluation strategy* [23]. The most famous evaluation strategies are *call-by-value*, that 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* [25, 4], which extends call-by-name with a memoization or sharing mechanism to remember the value of an argument that has already been evaluated.

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

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

The first known abstract machine computing the normal form of a term is due to Crégut [14] 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 [16, 9, 2], call-by-name [1] or call-by-need [8, 10]. 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. As such 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. However, since call-by-need has been shown to capture optimal weak reduction [7], it is known that the deliberate use of strong reduction [17] 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 reduces only needed redexes. Second, it keeps a level of sharing at least equal to that of call-by-value and call-by-need. Third, it enables 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.

Neededness of a redex is undecidable in general, thus the first and third guidelines are antagonist. Section 3 resolves this tension by exposing a simple syntactic criterion capturing more needed redexes than what is already used in call-by-need strategies. Through reducing only needed redexes, our calculus enjoys a normalization preservation theorem that is stronger than usual: Any λ-term that is *weakly* normalizing in the pure λ-calculus (there is at least one reduction sequence to a normal form, but some other sequences may diverge) will be *strongly* normalizing in our calculus (any reduction sequence is normalizing). This strong normalization theorem, related to the usual *completeness* results of call-by-name or call-by-need strategies, is completely dealt with using a system of non-idempotent intersection types. This allows us to avoid the traditional tedious syntactic commutation lemmas to provide more elegant proofs. Our proof improves the technique used in [19, 8].

While our calculus contains the strong call-by-need strategy introduced in [8], it also allows more liberal call-by-need strategies that anticipate some strong reduction steps in order to reduce the overall length of the reduction to normal form. Section 4 presents a restriction of the calculus that guarantees reduction sequences of minimal length and in particular builds shorter reduction sequences

Finally, Section 5 presents a formalization of parts of our results in Abella [5]. Beyond the proof safety provided by such a tool, this formalization has also influenced the design of our strong call-by-need calculus itself in a positive way. In particular, it promoted a presentation based on SOS-style local reduction rules [24], which later became a lever for a more efficient use of non-idempotent intersection types. The formalization can be found at the following address: `https://www.lri.fr/~blsk/SCBNd-FSCD2021.zip`

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 [8] 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. It includes however no 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 [4].

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

The free variables $\mathsf{fv}(t)$ of a term t are defined as usual. We write $\mathcal{C}$ for a context, that is a term with exactly one hole $\Box$, and L for a context with the specific shape $\Box[x_1\backslash t_1]\dots[x_n\backslash t_n]$. We write $\mathcal{C}[t]$ for the term obtained by plugging the subterm t in the hole of the context $\mathcal{C}$ (with possible capture of free variables of t by binders in $\mathcal{C}$), or tL when the context is of the specific shape L. We also write $\mathcal{C}[\![t]\!]$ for plugging a term t whose free variables are not captured by $\mathcal{C}$. The *values* we consider are λ-abstractions.

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 [18], it builds on the *Linear Substitution Calculus* [22, 3] that is more similar to the let-constructs commonly used for defining call-by-need.

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

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 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, that is 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. Remark that this calculus only allows the substitution of λ-abstractions, and not of variables as is it sometimes seen [21]. 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.

A term t of λ_{c} 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.

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

Through unfolding, any reduction step $t \to_{\mathsf{c}} u$ in λ_{c} 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 [13, 15] originally defined in [20]. A type τ may be a type variable α or an arrow type $\mathcal{M} \to \tau$, where $\mathcal{M}$ is a multiset $\{\!\{\sigma_1, \dots, \sigma_n\}\!\}$ 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. A typing judgment $\Gamma \vdash t : \tau$ assigns exactly one type to the term t. As shown by the typing rules in Fig. 1, an argument of an application or of an explicit substitution may be typed several times in a derivation. 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. Positive type occurrences $\mathcal{T}_+(\Gamma \vdash t : \tau)$ and negative type occurrences $\mathcal{T}_-(\Gamma \vdash t : \tau)$ of a typing judgment are defined by the following equations.

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

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

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

Figure 1 Typing rules for λ_{c}

▶ **Theorem 1** (Typability [8]). *If the pure λ-term t is weakly normalizing for β-reduction, then there is a typing judgment $\Gamma \vdash t : \tau$ such that $\{\!\{\}\!\} \notin \mathcal{T}_+(\Gamma \vdash t : \tau)$.*

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 for which the derivation Φ contains a typing judgment. 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.

▶ **Theorem 2** (Weighted subject reduction [8]). *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.*

3 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 needed redexes are reduced. Notice that gc-reduction is never needed in this calculus and will thus be ignored from now on.

3.1 Structures and frozen variables

The starting point is the same as the one for the original (weak) call-by-need calculus. Since the argument of a function is not always needed, we do not reduce in advance the right part of an application $t\ u$. Instead, we first evaluate t to an answer $(\lambda x.t')L$, then apply a β-reduction to put the argument u in the environment of t', and then go on with the resulting term $t'[x\backslash u]L$, evaluating u only if and when it is required.

The previous principle is enough for weak reduction, but new behaviors appear with strong reduction. Suppose for instance that we get an abstraction $\lambda x.x\ t\ u$ at top-level. Given its position, 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 [8], we call this variable x *frozen*. Now regarding the arguments t and u given to the frozen variable x, these will always remain at their respective positions and never get into contact with other parts of the term; they are in *top-level-like* positions and can be treated as independent terms. In particular, if t is an abstraction $\lambda y.t'$ then the variable y will never be substituted and can be seen as frozen.

We call *structure* an application $x\ t_1\ \ldots\ t_n$ led by a frozen variable x, possibly interlaced with explicit substitutions. As implied by the last rule in Fig. 2, an explicit substitution in a structure may even affect the leading variable, provided that the content of the substitution is itself a structure. We write $\mathcal{S}_\varphi$ the set of structures under a set φ of frozen variables. It differs from the notion in [8] in that it does not require the term to be in normal form.

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

Figure 2 Structures of λ_{sn}.

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

3.2 Reduction in λ_{sn}

As a result, the reduction relation of our calculus has three parameters:

- a rule ρ, which can be dB, lsv, or others that we will introduce shortly;
- a set φ of frozen variables, used to define structures and top-level-like positions;
- a flag μ, which is $\top$ for reduction at a top-level-like position, and $\bot$ for reduction at an arbitrary position.

The rules are given in Fig. 3. Rule @-LEFT makes reduction always possible on the left of an application, but as shown by the premise this position is not a $\top$ position. Rule @-RIGHT on the other hand shows that reduction on the right of an application is possible only in the context of a frozen structure. In this case however the position is $\top$.

Rules λ-TOP and λ-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.

Rules ES-LEFT and ES-LEFT-φ 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 position eligible for reduction. It uses an auxiliary rule id_x that propagates using the same inference rules as the rules dB and lsv, to probe a term for the presence of some variable x at a reduction position. By freshness, $x \notin \varphi$. This auxiliary rule does not modify the term to which it applies, as witnessed by its base case ID.

Rules DB and LSV are the base cases for applying dB- or lsv-reduction. Using the notations of λ_{c}, they will allow the following reductions.

$$\begin{array}{rcll} (\lambda x.t)L\ u & \xrightarrow{\mathsf{dB},\varphi,\mu}_{\mathsf{sn}} & t[x\backslash u]L & \\ \mathcal{C}[\![x]\!][x\backslash vL] & \xrightarrow{\mathsf{lsv},\varphi,\mu}_{\mathsf{sn}} & \mathcal{C}[\![v]\!][x\backslash v]L & \text{with } v \text{ value, and } \mathcal{C} \text{ a suitable context} \end{array}$$

Each is defined using an auxiliary reduction relation dealing with the list L of explicit substitutions. These auxiliary reductions are given in Fig. 4.

Rules DB-BASE and LSV-BASE describe the base cases of the auxiliary reductions, where the list L is empty. Note that, while DB-BASE is an axiom, the inference rule LSV-BASE uses as a premise a $\xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}}$-reduction using a new reduction rule $\mathsf{sub}_{x\backslash v}$. This reduction rule substitutes one occurrence of the variable x at a position eligible for reduction by the value v (with, by freshness, $x \notin \varphi$). As seen for id_x above, this reduction rule propagates using the same inference rules as dB and lsv, and its base case is the rule SUB in Fig. 3. The presence

$$\text{@-LEFT}\ \frac{t \xrightarrow{\rho,\varphi,\bot}_{\mathsf{sn}} t'}{t\ u \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'\ u} \qquad \text{@-RIGHT}\ \frac{t \in \mathcal{S}_\varphi \qquad u \xrightarrow{\rho,\varphi,\top}_{\mathsf{sn}} u'}{t\ u \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t\ u'} \qquad \lambda\text{-TOP}\ \frac{t \xrightarrow{\rho,\varphi\cup\{x\},\top}_{\mathsf{sn}} t'}{\lambda x.t \xrightarrow{\rho,\varphi,\top}_{\mathsf{sn}} \lambda x.t'} \qquad \lambda\text{-BOT}\ \frac{t \xrightarrow{\rho,\varphi,\bot}_{\mathsf{sn}} t'}{\lambda x.t \xrightarrow{\rho,\varphi,\bot}_{\mathsf{sn}} \lambda x.t'}$$

$$\text{ES-LEFT}\ \frac{t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'}{t[x\backslash u] \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'[x\backslash u]} \qquad \text{ES-LEFT-}\varphi\ \frac{t \xrightarrow{\rho,\varphi\cup\{x\},\mu}_{\mathsf{sn}} t' \qquad u \in \mathcal{S}_\varphi}{t[x\backslash u] \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'[x\backslash u]} \qquad \text{ES-RIGHT}\ \frac{t \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}} t \qquad u \xrightarrow{\rho,\varphi,\bot}_{\mathsf{sn}} u'}{t[x\backslash u] \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t[x\backslash u']}$$

$$\text{ID}\ \frac{}{x \xrightarrow{\mathsf{id}_x,\varphi,\mu}_{\mathsf{sn}} x} \qquad \text{SUB}\ \frac{}{x \xrightarrow{\mathsf{sub}_{x\backslash v},\varphi,\mu}_{\mathsf{sn}} v} \qquad \text{DB}\ \frac{t \to_{\mathsf{db}} t'}{t \xrightarrow{\mathsf{dB},\varphi,\mu}_{\mathsf{sn}} t'} \qquad \text{LSV}\ \frac{t \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t'}{t \xrightarrow{\mathsf{lsv},\varphi,\mu}_{\mathsf{sn}} t'}$$

Figure 3 Reduction rules for λ_{sn}.

$$\text{DB-BASE}\ \frac{}{(\lambda x.t)\ u \to_{\mathsf{db}} t[x\backslash u]} \qquad \text{LSV-BASE}\ \frac{t \xrightarrow{\mathsf{sub}_{x\backslash v},\varphi,\mu}_{\mathsf{sn}} t' \qquad v \text{ value}}{t[x\backslash v] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t'[x\backslash v]}$$

$$\text{DB-}\sigma\ \frac{t\ u \to_{\mathsf{db}} v}{t[x\backslash w]\ u \to_{\mathsf{db}} v[x\backslash w]} \qquad \text{LSV-}\sigma\ \frac{t[x\backslash u] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t'}{t[x\backslash u[y\backslash w]] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t'[y\backslash w]} \qquad \text{LSV-}\sigma\text{-}\varphi\ \frac{t[x\backslash u] \xrightarrow{\varphi\cup\{y\},\mu}_{\mathsf{lsv}} t' \qquad w \in \mathcal{S}_\varphi}{t[x\backslash u[y\backslash w]] \xrightarrow{\varphi,\mu}_{\mathsf{lsv}} t'[y\backslash w]}$$

Figure 4 Auxiliary reduction rules for λ_{sn}.

of this premise $t \xrightarrow{\mathsf{sub}_{x\backslash v},\varphi,\mu}_{\mathsf{sn}} t'$ in the rule is the primary reason why the auxiliary relation $\xrightarrow{\varphi,\mu}_{\mathsf{lsv}}$ is parameterized by φ and μ. The combination 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 [8], a similar effect was achieved using a more convoluted condition on a composition of contexts.

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.

Rules LSV-σ and LSV-σ-φ achieve the same effect with the nested substitutions applied to the value substituted by an lsv-reduction step. As with rule ES-LEFT-φ, if the substitution is a structure, the variable can be frozen. Note that this last feature distinguishing LSV-σ and LSV-σ-φ can be ignored for now. It will prove useful in Sec. 4 only.

Here is a sample derivation using our inference rules. Its left branch checks that an occurrence of the variable x is actually at a needed position, while its right branch reduces

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

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

Figure 5 Normal forms of λ_{sn}.

the argument of the substitution.

$$\dfrac{\dfrac{a \in \mathcal{S}_{\{a\}} \qquad \dfrac{}{x \xrightarrow{\mathsf{id}_x,\{a\},\top}_{\mathsf{sn}} x}\ \text{ID}}{a\ x \xrightarrow{\mathsf{id}_x,\{a\},\top}_{\mathsf{sn}} a\ x}\ @\text{-RIGHT}}{\lambda a.a\ x \xrightarrow{\mathsf{id}_x,\emptyset,\top}_{\mathsf{sn}} \lambda a.a\ x}\ \lambda\text{-TOP} \qquad \dfrac{\dfrac{\dfrac{}{(\lambda y.t)\ v \to_{\mathsf{db}} t[y\backslash v]}\ \text{DB-BASE}}{(\lambda y.t)[z\backslash u]\ v \to_{\mathsf{db}} t[y\backslash v][z\backslash u]}\ \text{DB-}\sigma}{(\lambda y.t)[z\backslash u]\ v \xrightarrow{\mathsf{dB},\emptyset,\bot}_{\mathsf{sn}} t[y\backslash v][z\backslash u]}\ \text{DB}}{(\lambda a.a\ x)[x\backslash(\lambda y.t)[z\backslash u]\ v] \xrightarrow{\mathsf{dB},\emptyset,\top}_{\mathsf{sn}} (\lambda a.a\ x)[x\backslash t[y\backslash v][z\backslash u]]}\ \text{ES-RIGHT}$$

The actual reduction, written $\to_{\mathsf{sn}}$, is either $\xrightarrow{\mathsf{dB},\varphi,\top}_{\mathsf{sn}}$ or $\xrightarrow{\mathsf{lsv},\varphi,\top}_{\mathsf{sn}}$, with φ typically being empty when reducing closed terms, or containing the free variables otherwise.

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

Proof. By induction on the reduction $t \xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}} t'$. Included in our Abella development. ◀

The normal forms of λ_{sn} correspond to the normal forms of the strong call-by-need strategy [8]. They can be characterized by the inductive definition given in Fig. 5.

▶ **Lemma 4.** *$t \in \mathcal{N}_\varphi$ if and only if there is no reduction $t \xrightarrow{\rho,\varphi,\mu}_{sn} t'$.*

Proof. The first part (a term cannot be in normal form and reducible) has been proved in Abella, 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 (not in Abella yet). ◀

▶ **Lemma 5.** *If $t \in \mathcal{N}_\varphi$ then $t^\star$ is a normal form in the λ-calculus.*

Proof. By induction on $t \in \mathcal{N}_\varphi$. Done in Abella. ◀

Finally, note that the strong call-by-need strategy introduced in [8] is included in our calculus. One can recover this strategy by imposing two restrictions on $\xrightarrow{\rho,\varphi,\mu}_{\mathsf{sn}}$:

- remove the rule λ-BOT, since the strategy never reduces under an abstraction that is not yet in top-level-like position,
- restrict the rule @-RIGHT to the case where the left member of an application is not only a structure, but rather a structure in normal form, since the strategy imposes left-to-right reduction of structures.

$$\text{TY-VAR}\ \frac{}{x : \{\!\{\sigma\}\!\} \vdash^{\mu}_{\varphi} x : \sigma} \qquad \text{TY-}\lambda\text{-}\bot\ \frac{\Gamma; x : \mathcal{M} \vdash^{\bot}_{\varphi} t : \tau}{\Gamma \vdash^{\bot}_{\varphi} \lambda x.t : \mathcal{M} \to \tau} \qquad \text{TY-}\lambda\text{-}\top\ \frac{\Gamma; x : \mathcal{M} \vdash^{\top}_{\varphi\cup\{x\}} t : \tau}{\Gamma \vdash^{\top}_{\varphi} \lambda x.t : \mathcal{M} \to \tau}$$

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

$$\text{TY-ES}\ \frac{\Gamma; x : \mathcal{M} \vdash^{\mu}_{\varphi} t : \tau \qquad (\Delta_\sigma \vdash^{\bot}_{\varphi} u : \sigma)_{\sigma\in\mathcal{M}}}{\Gamma + \sum_{\sigma\in\mathcal{M}} \Delta_\sigma \vdash^{\mu}_{\varphi} t[x\backslash u] : \tau} \qquad \text{TY-ES-}\mathcal{S}\ \frac{\Gamma; x : \mathcal{M} \vdash^{\mu}_{\varphi\cup\{x\}} t : \tau \qquad u \in \mathcal{S}_\varphi \qquad (\Delta_\sigma \vdash^{\bot}_{\varphi} u : \sigma)_{\sigma\in\mathcal{M}}}{\Gamma + \sum_{\sigma\in\mathcal{M}} \Delta_\sigma \vdash^{\mu}_{\varphi} t[x\backslash u] : \tau}$$

Figure 6 Annotated system for non-idempotent intersection types.

3.3 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 (Th. 11). The proof relies on the non-idempotent intersection type system in the following way:

1. Typability (Th. 1) ensures that any weakly normalizing λ-term admits a typing derivation (with no positive occurrence of $\{\!\{\}\!\}$).
2. We prove here that any λ_{sn}-reduction in a typed λ_{sn}-term t (with no positive occurrence of $\{\!\{\}\!\}$) is at a typed position of t (Th. 10).
3. Then weighted subject reduction (Th. 2) provides a decreasing measure for λ_{sn}-reduction.

Finally, the obtained normal form is related to the β-normal form using Lemmas 3, 4, and 5.

The proof of the forthcoming typed reduction (Th. 10) uses a refinement of the non-idempotent intersection types system of λ_{c}, given in Fig. 6. Both systems derive the same typing judgments with the same typed positions. The refined system however features an annotated typing judgment $\Gamma \vdash^{\mu}_{\varphi} t : \tau$ embedding additional context information, such as a set φ of frozen variables and a marker μ of top-level-like positions. These annotations are faithful counterparts to the corresponding annotations of λ_{sn} reduction rules.

In particular, the rule for typing an abstraction is split in two versions TY-λ-$\bot$ and TY-λ-$\top$, the latter being applicable to $\top$ positions and thus freezing the variable bound by the abstraction (in both rules, by freshness convention we assume $x \notin \varphi$). The rule for typing an application is also split in two version: TY-@-$\mathcal{S}$ is applicable when the left part of the application is a structure and marks the right part as a $\top$ position, while TY-@ is applicable otherwise. Finally, the rule for typing an explicit substitution is similarly split in two versions, depending on whether the contents of the substitution is a structure or not, and handling the set of frozen variables accordingly. We write $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t : \tau$ if there is a derivation Φ of the annotated typing judgment $\Gamma \vdash^{\mu}_{\varphi} t : \tau$.

▶ **Lemma 6** (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 erasing of the annotations, but is not used in the proof of the completeness result.

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

▶ **Lemma 7** (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.[1] 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. ◀

Given a derivation Φ, we write $\mathsf{fzt}(\Phi)$ the set of types associated to frozen variables in judgments of the derivation Φ.

▶ **Lemma 8** (Subformula property).

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

Proof. By mutual induction on the typing derivations.[1] Most cases are fairly straightforward. The only difficult case comes from the rule TY-@-$\mathcal{S}$, 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 (Lem. 7) to conclude. ◀

▶ **Lemma 9** (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 7 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 8 $\mathcal{M} \in \mathcal{T}_+(\Gamma \vdash^{\mu}_{\varphi} t : \tau)$, thus $\mathcal{M} \neq \{\!\{\}\!\}$. ◀

The main theorem on typed reduction is then as follows.

▶ **Theorem 10** (Typed reduction). *If* $\Phi \rhd \Gamma \vdash^{\mu}_{\varphi} t : \tau$ *with* $\{\!\{\}\!\} \notin \mathcal{T}_+(\Gamma \vdash t : \tau)$*, then every* λ_{sn}*-reduction* $t \xrightarrow{\rho,\varphi,\mu}_{sn} t'$ *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 9). Since all other reduction cases concern positions that are systematically typed, we focus here on @-RIGHT and ES-RIGHT.

- Case @-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.

[1] See appendix for the complete proof.

- 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 11** (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 $n^\star_{sn} = n_\beta$.*

Proof. Let t be a pure λ-term that admits a normal form n_β for β-reduction. By Theorem 1 there exists a derivable typing judgment $\Gamma \vdash t : \tau$ such that $\{\!\{\}\!\} \notin \mathcal{T}_+(\Gamma \vdash t : \tau)$. Thus by Theorems 10 and 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, $n_{\mathsf{sn}} \in \mathcal{N}_\varphi$, and by Lemma 5, $n^\star_{\mathsf{sn}}$ is a normal form in the λ-calculus. Moreover, by simulation (Lem. 3), there is a reduction $t^\star \to^*_\beta n^\star_{\mathsf{sn}}$. By uniqueness of the normal form in the λ-calculus, $n^\star_{\mathsf{sn}} = n_\beta$. ◀

4 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 will be used to restrict the *value* criterion in the LSV-BASE rule of Fig. 4. We then show that this restriction is strong enough to guarantee the diamond property. Finally, we explain why our restricted calculus, named $\lambda_{\mathsf{sn}+}$, only produces minimal sequences, among all the reduction sequences allowed by λ_{sn}. This makes it a relatively optimal strategy.

4.1 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 $id = \lambda x.x$. 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.

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

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

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

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

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

Figure 7 Local normal forms.

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

Figure 8 New rule LSV-BASE for $\lambda_{\mathsf{sn}+}$.

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

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

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 were in a $\top$ position. But in a $\bot$ position, variable y is not frozen, which prevents any further reduction of $y\ \Omega$. The inference rules are presented in Fig. 7.

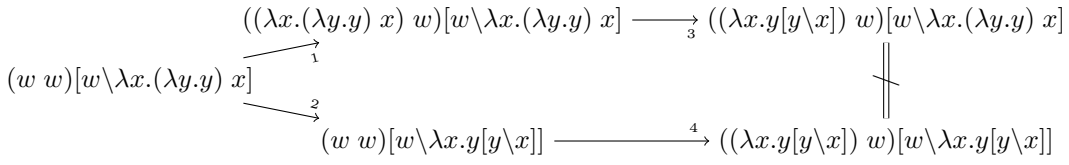
If an abstraction is in a $\top$ position, its variable is added to the set φ of frozen variables, as in Fig. 3. But if an abstraction is in a $\bot$ position, its variable is added to a new set ω, as shown in rule λ-ω of Fig. 7. That is what will happen to y in $\lambda y.y\ \Omega$.

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 @-ω of Fig. 7. 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.

Now that we have characterized the local normal forms. Let us revisit our λ_{sn}-calculus. The rules of $\lambda_{\mathsf{sn}+}$ are the same as the ones of λ_{sn} presented in Fig. 4 and Fig. 3, except that a new rule replaces LSV-BASE. This rule is shown in Fig. 8.

4.2 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. As a consequence, these calculi cannot have the diamond property, as shown on the following example.



In λ_{sn}, the leftmost term can be reduced, either by rule lsv (arrow 1) because the substituted term is a value, or by rule dB (arrow 2). The top term can only be reduced by rule dB (arrow 3) because the substitution variable is not reachable. The bottom term can only be reduced by rule lsv (arrow 4) because the substituted term is not reducible. The two new terms are different, thus breaking the diamond property. It would take one more reduction step (in λ_{c}) for the top sequence to reach the bottom-right term. But in our restricted calculus $\lambda_{\mathsf{sn}+}$, arrow 1 is forbidden, since the substituted term is not in local normal form. By preventing such sequences, the diamond property is restored.

▶ **Theorem 12** (Diamond). *Suppose* $t \xrightarrow{\rho_1,\varphi,\mu}_{sn+} t_1$ *and* $t \xrightarrow{\rho_2,\varphi,\mu}_{sn+} t_2$. *Assume that, if* ρ_1 *and* ρ_2 *are* sub *or* id, *then they apply to separate variables. Then there exists* t' *such that* $t_1 \xrightarrow{\rho_2,\varphi,\mu}_{sn+} t'$ *and* $t_2 \xrightarrow{\rho_1,\varphi,\mu}_{sn+} t'$.

Proof. The statement 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 $\xrightarrow{\varphi,\mu}_{\mathsf{lsv}}$. Then it becomes a tedious but rather unsurprising induction on t, with reasoning by case on the last inference rule applied on each side. One notable case is when the two reductions are respectively given by rules @-LEFT and @-RIGHT. Indeed, the reduction on the left does not interfere with the reduction on the right thanks to a stability property of structures (Lem. 13 below). ◀

▶ **Lemma 13** (Stability of structures). *If* $t \in \mathcal{S}_\varphi$ *and* $t \xrightarrow{\rho,\varphi,\mu}_{sn+} t'$ *then* $t' \in \mathcal{S}_\varphi$

4.3 Minimal path length

The $\lambda_{\mathsf{sn}+}$-calculus is a restriction of λ_{sn} that causes term to be reduced to local normal form before they can be substituted (Fig. 8), thus reducing the amount of duplicated work. Except for this restriction, any reduction sequence of λ_{sn} is admissible in $\lambda_{\mathsf{sn}+}$. So, any λ_{sn}-reduction sequence to a normal form can be turned into a $\lambda_{\mathsf{sn}+}$ one that is no longer. Since the diamond property ensures that all the sequences to a normal form have the same length in $\lambda_{\mathsf{sn}+}$, this guarantees that the reduction sequences are of minimal length.

▶ **Theorem 14** (Minimal length). *With* $t' \in \mathcal{N}_\varphi$, *if* $t \to^n_{sn} t'$ *and* $t \to^m_{sn+} t'$ *then* $m \leq n$.

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.

5 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 [11] and parametric higher-order abstract

syntax [12]. 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 [5], in the hope that it would make syntactic proofs more straightforward. This section describes our experience with this tool.[2]

5.1 Nominal variables and 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 in normal form for our calculus. The second line states that any nominal variable is in normal form, while the third line states that an abstraction is in normal form, as long as the abstracted term is in normal form for any nominal variable.

```
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` can only be a structure with respect to a variable that occurs in it. Had the quantifiers been exchanged, the statement would have been fine. Unfortunately, Abella kind of requires universal quantifiers to happen before nominal ones in theorem statements, thus exacerbating the issue. The correct way to state the above proposition is by carefully lifting any term in which a given free variable could occur:

```
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 needed to state that the free variable `x` occurring in `T` could not occur in `U`, by virtue of `star`. This was expressed as follows (with `y\V` denoting $y \mapsto V$):

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

[2] See appendix for all the definitions and the statement of the main theorems.

5.2 Judgments, contexts, and derivations

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

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

```
nf X :- frozen X.
nf (abs U) :- pi x\ frozen 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)}`. If there were only the two rules above, there would be only three ways of deriving the proposition. Indeed, it can be derived from `{L |- 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.*, 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 Fig. 2 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.

5.3 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. 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`.

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

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

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

Since this is just a relation, we have to prove that it is defined over all the closed terms of our calculus, that it maps only to pure 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 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 [3]. 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 [8]. 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 [8].

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 there is a different yet simple criterion that characterizes more needed redexes, and thus potentially allow for even shorter sequences than our calculus.

Even with the current criterion, there is still work to be done. First and foremost, the Abella formalization should be completed to at least include the diamond property. There are also some potential improvements to consider. For example, our calculus could be made to not substitute variables that are not applied (rule LSV-BASE), following [26, 2] 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, wich avoids more duplications but whose properties are tightly related to weak reduction [6]. Finally, this paper stops at describing the reduction rules of our calculus and does not investigate what an efficient abstract machine would look like.

References

1. 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.
2. Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. Strong call-by-value is reasonable, implosively, February 2021. arXiv:2102.06928.
3. 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.
4. 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*, POPL '95, pages 233–246, 1995. doi:10.1145/199448.199507.
5. 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.
6. 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*, POPL, pages 469–480, January 2012. doi:10.1145/2103656.2103713.
7. Thibaut Balabonski. Weak optimality, and the meaning of sharing. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming*, ICFP'13, pages 263–274, September 2013. doi:10.1145/2500365.2500606.
8. 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.
9. Malgorzata Biernacka, Dariusz Biernacki, Witold Charatonik, and Tomasz Drab. An abstract machine for strong call by value. September 2020. arXiv:2009.06984.
10. 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 (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:20, 2019. doi:10.4230/LIPIcs.FSCD.2019.8.
11. Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, October 2012. doi:10.1007/s10817-011-9225-2.
12. Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *13th ACM SIGPLAN International Conference on Functional Programming*, ICFP, pages 143–156, September 2008. doi:10.1145/1411204.1411226.
13. 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.
14. Pierre Crégut. An abstract machine for lambda-terms normalization. In *ACM Conference on LISP and Functional Programming*, LFP '90, page 333–340, 1990. doi:10.1145/91556.91681.
15. 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.
16. Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *7th ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, page 235–246, 2002. doi:10.1145/581478.581501.
17. Carsten Kehler Holst and Darsten Krogh Gomard. Partial evaluation is fuller laziness. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '91, page 223–233, 1991. doi:10.1145/115865.115890.
18. 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.
19. 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.

20 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.

21 John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, May 1998. doi:10.1017/S0956796898003037.

22 Robin Milner. Local bigraphs and confluence: Two conjectures. *Electron. Notes Theor. Comput. Sci.*, 175(3):65–73, June 2007. doi:10.1016/j.entcs.2006.07.035.

23 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.

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

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

26 Nobuko Yoshida. Optimal reduction in weak-lambda-calculus with shared environments. In *Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, page 243–252, 1993. doi:10.1145/165180.165217.

A Formal definitions

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

```
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\ 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 B U U'.
step R B (es T U) (es T' U) :- pi 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 B U U'.

step (idx X) B X X :- active X.
step (sub X V) B X V :- active X.
step db B T T' :- step_db T T'.
step lsv B T T' :- step_lsv B T T'.
```

A small difference with the core of the paper is the predicate `active`, which characterizes the variable being considered id_x (`idx`) and $\mathsf{sub}_{x\backslash v}$ (`sub`). This predicate is just a cheap way of remembering that the active variable is fresh yet not frozen.

Another difference is rule λ-BOT. While the antecedent of the rule is at position $\bot$ as in the paper, the consequent is in any position rather than just $\bot$. Since any term reducible in position $\bot$ is provably reducible in position $\top$, this is just a conservative generalization of the rule.

The auxiliary rules for $\lambda_{\mathsf{sn}+}$, as given in Fig. 4 and Fig. 8 for rule LSV-BASE, are the same as in the core of the paper.

```
step_db (app (abs T) U) (es T U).
step_db (app (es T W) U) (es T' W) :- pi x\ step_db (app (T x) U) (T' x).

step_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).
```

```
step_lsv B (es T (es U W)) (es T' W) :-
  pi x\ step_lsv B (es T (U x)) (T' x).
step_lsv B (es T (es U W)) (es T' W) :-
  pi x\ frozen x => step_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'.
```

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

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

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

```
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 Fig. 7 are as follows. As for the `step` relation, one of the rules for abstraction was generalized a bit with respect to the paper. This time, it is for the $\top$ position, since any term that is locally normal in a $\top$ position is also locally normal in a $\bot$ 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 B 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.
```

Normal forms of the λ-calculus are defined as follows:

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

Finally, let us remind the definitions of a pure λ-term, of a β-reduction, of a sequence of zero or more β-reductions, and of the unfolding operation from λ_{c} to λ.

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

beta (app M N) (app M' N) :- beta M M'.
beta (app M N) (app M N') :- beta N N'.
beta (abs R) (abs R') :- pi x\ beta (R x) (R' x).
beta (app (abs R) M) (R M).

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

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).
```

B Formally verified properties

This appendix states the theorems that were fully proved using Abella. First comes the simulation property (Lem. 3), which states that, if $T \to_{\mathsf{sn}+} U$, 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 not reducible in $\lambda_{\mathsf{sn}+}$ (part of Lemma 4).

```
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.
```

Finally, if T is a normal form of λ_{sn}, then $T^\star$ is a normal form of the λ-calculus (Lem. 5).

```
Theorem correctness1' : forall T T*,
  {nf T} -> {star T T*} -> {nfb T*}.
```

C Proof of the subformula properties

We recall here Lemma 7:
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.

- 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-@ and TY-@-$\mathcal{S}$. Both have a premise $\Gamma' \vdash^\bot_\varphi t_1 : \mathcal{M} \to \tau$ 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.

- Case where $t_1[x\backslash t_2] \in \mathcal{S}_\varphi$ 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 $y \in \mathsf{dom}(\Gamma'')$ and $y \in \mathsf{dom}(\Gamma)$, 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 $\Delta \vdash^\bot_\varphi t_2 : \sigma$ 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 $\tau \in \Gamma$. ◀

We recall here Lemma 8:

1. If $\Phi \rhd \Gamma \vdash^\top_\varphi t : \tau$ then $\begin{cases} \mathcal{T}_+(\mathsf{fzt}(\Phi)) & \subseteq & \bigcup_{x\in\varphi} \mathcal{T}_+(\Gamma(x)) \cup \mathcal{T}_-(\tau) \\ \mathcal{T}_-(\mathsf{fzt}(\Phi)) & \subseteq & \bigcup_{x\in\varphi} \mathcal{T}_-(\Gamma(x)) \cup \mathcal{T}_+(\tau) \end{cases}$
2. If $\Phi \rhd \Gamma \vdash^\bot_\varphi t : \tau$ then $\begin{cases} \mathcal{T}_+(\mathsf{fzt}(\Phi)) & \subseteq & \bigcup_{x\in\varphi} \mathcal{T}_+(\Gamma(x)) \\ \mathcal{T}_-(\mathsf{fzt}(\Phi)) & \subseteq & \bigcup_{x\in\varphi} \mathcal{T}_-(\Gamma(x)) \end{cases}$

Proof. By mutual induction on the typing derivations.

- Both properties are immediate in case TY-VAR, where $\mathsf{fzt}(\Phi) = \{\sigma\}$.
- Cases for abstractions.
  - If $\Phi \rhd \Gamma \vdash^\bot_\varphi \lambda x.t : \mathcal{M} \to \tau$ by rule TY-λ-$\bot$ with premise $\Phi' \rhd \Gamma; x : \mathcal{M} \vdash^\bot_\varphi t : \tau$. 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-λ-$\top$ 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-@ are by immediate application of the induction hypotheses.
  - If $\Phi \rhd \Gamma \vdash^\mu_\varphi t\ u : \tau$ by rule TY-@-$\mathcal{S}$, with premises $\Phi_t \rhd \Gamma_t \vdash^\bot_\varphi t : \mathcal{M} \to \tau$, $t \in \mathcal{S}_\varphi$ 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.
    By immediate use of 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 substitutions are by immediate application of the induction hypothesis. ◀