Pattern Matching and Fixed Points: Resource Types and Strong Call-By-Need

Extended Abstract

Pablo Barenbaum  
Universidad de Buenos Aires and  
CONICET  
foones@gmail.com

Eduardo Bonelli  
Stevens Institute of Technology  
eabonelli@gmail.com

Kareem Mohamed  
Stevens Institute of Technology  
KareemMohamed@outlook.com

ABSTRACT

Resource types are types that statically quantify some aspect of program execution. They come in various guises; this paper focusses on a manifestation of resource types known as non-idempotent intersection types. We use them to characterize weak normalisation for a type-erased lambda calculus for the Calculus of Inductive Construction (\(\lambda e\)), as introduced by Gregoire and Leroy. The \(\lambda e\) calculus consists of the lambda calculus together with constructors, pattern matching and a fixed-point operator. The characterization is then used to prove the completeness of a strong call-by-need strategy for \(\lambda e\). This strategy operates on open terms: rather than having evaluation stop when it reaches an abstraction, as in weak call-by-need, it computes strong normal forms by admitting reduction inside the body of abstractions and substitutions. Moreover, argument evaluation is by-need: arguments are evaluated when needed and at most once. Such a notion of reduction is of interest in areas such as partial evaluation and proof-checkers such as Coq.

ACM Reference Format:

1 INTRODUCTION

This paper is about the lambda calculus, the programming language that lies at the core of all Functional Programming Languages (FPL). FPLs evaluate programs until a value is obtained, if such a value exists at all. Programs are modeled as closed lambda calculus terms; values are a subset of programs that represent the possible results that one obtains by evaluation, typical examples being numerals, booleans and abstractions. FPLs implement an evaluation strategy called weak reduction since evaluation does not take place under an abstraction. When computing values from programs, such strategies typically also implement some form of memoization or sharing in order to avoid performing duplicate work. Moreover, this work is only performed if it is actually needed for obtaining a value. One thus speaks of call-by-need strategies for weak reduction, a notion that originates in the seminal work of Wadsworth [27].

Often one is interested in reducing inside the bodies of abstractions. One simple example is a technique known as partial evaluation (PE). In PE one has knowledge about some, but not all, of the arguments to a function, the remaining ones being supplied at a later stage. In this case, one can specialize the code of the function to those specific arguments. Here is a classic example. Assume we have a function for computing \(m^n\), \(n \geq 0\):

\[
pow := \lambda n.\lambda m.\text{if } n = 0 \text{ then } 1 \text{ else } m \ast pow(n - 1) m
\]

If we know the value of \(n\) to be 2, then we can produce a more efficient version of \(\text{pow}\) 2 as follows:

\[
pow 2 \rightarrow \lambda m.\text{if } 2 = 0 \text{ then } 1 \text{ else } m \ast (\text{pow} (2 - 1) m)
\]

\[
\rightarrow \lambda m. m \ast (\text{pow} (2 - 1) m)
\]

\[
\rightarrow \lambda m. m \ast (\text{if } 1 = 0 \text{ then } 1 \text{ else } m \ast (\text{pow} (1 - 1) m))
\]

\[
\rightarrow \lambda m. m \ast (\text{if } 0 = 0 \text{ then } 1 \text{ else } m \ast (\text{pow} (0 - 1) m))
\]

\[
\rightarrow \lambda m. m \ast (m + 1)
\]

\[
\rightarrow \lambda m. m \ast m
\]

Notice that all the reduction steps depicted above, take place under the lambda abstraction \(\lambda m\). Such a notion of reduction is called strong reduction. The values computed are normal forms which we refer to as strong normal forms to distinguish them from the normal forms of weak reduction.

Another area of interest of strong reduction is in the implementation of proof assistants that require checking for definitional equality. Proof assistants, such as Coq, that rely on definitional equality of types typically include a typing rule called conversion:

\[
\Gamma \vdash t : \tau \quad \tau \equiv \sigma
\]

\[
\Gamma \vdash t : \sigma
\]

Checking that types \(\tau\) and \(\sigma\) are definitionally equal, denoted by the judgement \(\tau \equiv \sigma\), involves computing the strong normal form of these types. In turn, this involves computing the strong normal form of the terms that occur in them. The reason that terms occur in types is that the type theory on which such proof assistants are erected are dependent type theories. These terms include constants for building values of inductive types and fixed-point operators for encoding recursive functions over inductive types.
The Extended Lambda Calculus. The Extended Lambda Calculus [21] (referred in op.cit. as the "type-erased lambda calculus"), denoted $\lambda e$, extends the lambda calculus with constants, pattern matching and fixed-points. Here is an example of a term in $\lambda e$ that computes the length of a list encoded with constants nil and cons (see Sec. 2 for a detailed definition of $\lambda e$):

$$f \cdot (\lambda x. \text{case } x \equiv \text{nil } \Rightarrow \text{zero } \cdot \text{cons } \text{hd } t \Rightarrow \text{suc } (l t))$$

The Extended Lambda Calculus is a subset of Gallina, the specification language of Coq. Gregoire and Leroy [21] study judicious mechanisms for implementing strong reduction in $\lambda e$ in order to apply it to check type-conversion. They propose a notion of strong reduction for $\lambda e$ on open terms (i.e. terms possibly containing free variables) called symbolic call-by-value. Symbolic CBV iterates call-by-value, accumulating terms for which computation cannot progress. No notion of sharing is addressed. Indeed, unnecessary computation may be performed. For example, consider the following $\lambda e$ term, where $\text{id}$ abbreviates the identity term $\lambda z. z$:

$$\text{case } e \equiv \text{id } \text{id} \Rightarrow d$$

(1)

This term is a case expression that has condition $e \equiv \text{id } \text{id}$ and branch $e \Rightarrow d$, the pattern of the branch being $e$ and the target $d$. Notice that the branch does not make use of $x$ in the target. However, Symbolic CBV will contract the redex $\text{id } \text{id}$ since the argument of $e$ must be a value before selecting the matching branch.

Adding Sharing to Strong Reduction. This paper proposes a notion of strong reduction for $\lambda e$ that only reduces redexes that are needed in order to obtain the (strong) normal form. e.g. our strategy will not reduce the $\text{id } \text{id}$ redex in (1). Arguments to functions will be suspended until needed. Moreover, they will be reduced at most once. The latter requires us to extend $\lambda e$ with an additional syntactic construct to hold such suspended arguments: explicit substitutions. The resulting theory of sharing ($\lambda_{sh}$) replaces the usual $\beta$ rule in the lambda calculus $(\lambda x. t) s \rightarrow_{\beta} t[x := s]$ with $(\lambda x. t) s \rightarrow_{sh} t[x := s]$. The term $t[x := s]$ is also often written let $x = s$ in $t$. Notice that rather than substitute all free occurrences of $x$ in $t$ with $s$, the latter suspends this substitution process. Moreover, since explicit substitutions may now hide redexes, such as in $(\lambda x. x)[\lambda y. z]$, a slightly more general formulation of $\lambda_{sh}$ is adopted, namely $(\lambda x. t) L s \rightarrow_{sh} t[x := s][L]$. The notation $L$ denotes a possibly empty list of explicit substitutions [6].

In order to make use of an argument suspended in an explicit substitution it has to have been fully evaluated to a result. As mentioned above, results typically include numerals, booleans and abstractions. In our setting, values shall either be abstractions or terms headed with constants (cf. Sec. 2.1). An additional consideration is that values $v$ may be "polluted" with explicit substitutions $L$. We thus have the following rule to be able to use a suspended argument: $C[x]\|\|L \rightarrow_{1sv} C[v]\|\|L$. Note how this rule makes use of a context $C$ and the notation $C[x]\|\|$ to mean that there is a free occurrence of $x$. For example, $(x x)[\lambda y. y] \rightarrow_{1sv} (\lambda y. y) x[x := y y]$ and also $(x x)[\lambda y. y] \rightarrow_{1sv} (\lambda y. y) x[x := y y]$. Of course, also $\lambda y. z) x[x := y y] \rightarrow_{1sv} ((\lambda y. z) y) x[x := y y]$, even though $x$ is not needed since it will be discarded by $(\lambda y. z)$. Selecting only needed occurrences of $x$ to be replaced by results will be achieved by imposing a specific reduction strategy on $\rightarrow_{sh}$, as described below.

Additional rules for dealing with case expressions and fixed-points are discussed in Sec. 2.1.

Resource Types for the Lambda Calculus. The challenge in establishing that the strong call-by-need theory is well-behaved is proving that every term in $\lambda e$ that has a normal form also has a normal form in $\lambda_{sh}$ with sharing ($\lambda_{sh}$). That is, that the restricted notion of replacement of results is general enough to capture all normalising derivations in $\lambda e$.

$$t \in WN(\lambda e) \Rightarrow t \in WN(\lambda_{sh})$$

The notation $\text{WN}(\lambda e)$ denotes the set of $\lambda e$-terms that have a normal form via $\lambda e$ and similarly for $\text{WN}(\lambda_{sh})$. Arguably this has been the main technical hurdle in prior works for weak reduction such as [8, 25] which introduced ad hoc notions of development, reduction tracking and dags with boxes. It was recently noticed [23] that by devising an appropriate non-idempotent intersection type system $\mathcal{T}$ for $\lambda_{sh}$, one could achieve this as follows:

\begin{align*}
\text{Step 1} & \quad t \in W N(\lambda e) \quad \Rightarrow \quad t \in \text{Typable}(\mathcal{T}) \\
\text{Step 2} & \quad t \in \text{Typable}(\mathcal{T}) \quad \Rightarrow \quad t \in W N(\lambda_{sh})
\end{align*}

Non-idempotent intersection types [20] track/count the uses of variables in terms and thus restrict reduction properties of its typable terms e.g. they may be used to characterize weak, head and strongly normalising terms [12]. If one writes non-idempotent intersection types as multisets of types, then $x : [\tau_1, \tau_2]$ means that $x$ has to be used twice with the indicated types. Similarly, $y : [[\tau_1, \tau_2] \rightarrow \tau_3]$ means that $y$ has to be used once and that the argument to $y$ has to be typed twice, once with type $\tau_1$ and once with $\tau_2$.

The argument behind Step 1 is roughly as follows. Given a term in normal form, for any variable $x$, one “counts” each of its occurrences by giving it a type and then takes the multiset of all those types as the type of $x$. Then one shows a Subject Expansion result: if the contractum via a reduction step of a term is typable, then so is the term itself. For those variables that reduction does not erase, their type in the contractum can be used to type the redex, those that are erased are not typed at all in the redex (they occur in subterms that are typed with the empty multiset).

The argument behind Step 2 involves showing that reduction of redexes that are typed in $\mathcal{T}$ decreases the size of the type derivation. Reduction of redexes that are not typed could lead to non-termination [9, 23]. For example, $x \Omega$, where $\Omega$ is $\lambda x. x x$, is typable by setting $x$ to have type $\| \rightarrow a$, for $a$ a type variable; the empty multiset of type $\| \|$ allowing the typing of $\Omega$ to not be accounted for. However, the term is not normalising in $\lambda_{sh}$ or any theory of sharing that allows $\beta$ to be simulated.

Resource Types for the Extended Lambda Calculus. We must adapt this counting technique to the setting of case and fixed points. It turns out that the challenge lies in dealing with case (however, see Rem. 1). Consider the term:

$$\text{case } e \equiv (c \Rightarrow d) \cdot (d \Rightarrow \Omega)$$

It will evaluate to $d$ and hence should be typable in $\mathcal{T}$ (cf. Step 1). Since $\Omega$ does not participate at all in computing $d$, there is no need for $\mathcal{T}$ to account for it. Thus our proposed typing rules will only type branches that are actually used to compute the normal form. This, however, beckons the question of what happens with case
expressions that block. In an expression such as:
\[ \text{case } c \text{ of } (d \Rightarrow d) \cdot (e \Rightarrow e) \]
all its subexpressions are part of the normal form and hence should be typed. Our proposed typing rule shall ensure this, thus avoiding typing terms such as:
\[ \text{case } c \text{ of } (d \Rightarrow d) \cdot (e \Rightarrow \Omega) \]
where, although matching is blocked, have not strong normal form \( \lambda \).

Since blocked case expressions could be applied to arguments, further considerations are required. Consider the term:
\[ \text{(case } c \text{ of } d \Rightarrow d) \Omega \]
It does not have a normal form in \( \lambda_e \) or \( \lambda_{sh} \) and hence should not be typable (Step 2). To ensure that, we need the type assigned to this term to provide access to the types of the arguments to which it is applied, namely \( \Omega \), so that constraints on these types may be placed. In other words, we need to devise \( T \) such that it gives case \( c \text{ of } (d \Rightarrow d) \Omega \) a type that includes that of \( \Omega \). This would enable us to state conditions that do not allow this term to be typable but do allow a term such as (case \( c \text{ of } d \Rightarrow d \cdot e \) to be typed. This motivates our notions of error type and error log (cf. Sec. 4).

The above examples were all closed terms. Open terms pose additional problems. Consider the term:
\[ \text{case } x \text{ of } (c \Rightarrow d) \cdot (e \Rightarrow \Omega) \]
Although it does not have a normal form in \( \lambda_{sh} \), it is typable with type \( d \), if \( x : [c] \). Notice, moreover, that the empty multimset of types does not occur in the type of \( x \) (in fact, it meets all the requirements of [9, 23]). The reason it is typable is that \( \Omega \) is never accounted: since \( x \) is known to have type \([c]\), only the \( c \Rightarrow d \) branch is typed. Hence some restrictions on the types of free variables in Step 2 must be put forward, clearly variables cannot be assigned any type. In particular, it seems we should not allow constant types such as \( c \) to occur positively in the types of free variables. Indeed, we will require that constant types do not occur positively in the typing context and negatively in error logs and in the predicate (cf. notion of good typing judgements in Sec. 4.1). Note that constants can occur negatively in the types of variables. This allows terms such as \( x \cdot e \) to be typable.

One final consideration is that collecting all the requirements, both on empty multiset types and type constants, should still allow weakly normalising terms in \( \lambda_e \) to be typable in \( T \). We will see that this will indeed be the case.

A Strong Call-by-Need Strategy. As mentioned, reduction in the theory of sharing may involve reducing redexes that are not needed. By restricting reduction in \( \rightarrow_{sh} \) to a subset of the contexts where reduction can take place, we can ensure that only needed redexes are reduced. We next illustrate, through an example, our call-by-need strategy. The strategy will be denoted \( \rightarrow_{sh} \) "sh" for sharing. Consider the term:
\[ \text{(case } (\lambda y. x y)(\overline{\text{id}} \cdot \overline{\text{id}}) \text{ of } c \Rightarrow d) \cdot (\overline{\text{id}} \cdot \overline{\text{e}}) \]
It consists of a case expression applied to an argument. This case expression has a condition \( (\lambda y. x y)(\overline{\text{id}} \cdot \overline{\text{id}}) \), a branch \( c \Rightarrow d \) with pattern \( e \) and target \( d \), and is applied to an argument \( \overline{\text{id}} \cdot \overline{\text{e}} \). The first reduction step for this term is the same as for weak call-by-need, namely reducing the \( \beta \)-redex \( (\lambda y. x y)(\overline{\text{id}} \cdot \overline{\text{id}}) \) in the condition of the case. It must be reduced in order to determine which branch, if any, is to be selected. This \( \beta \)-redex is turned into \( (x y)[y](\overline{\text{id}} \cdot \overline{\text{id}} \cdot \overline{\text{e}}) \). The resulting term is:
\[ (\text{case } (x y)(\overline{\text{y}} \cdot \overline{\text{id}} \cdot \overline{\text{id}}) \text{ of } c \Rightarrow d)(\overline{\text{id}} \cdot \overline{\text{e}}) \]
A weak call-by-need strategy would stop there, since the case expression is stuck. In the strong case, however, reduction should continue to complete the evaluation of the term until a strong normal form is reached. Both the body of the explicit substitution \( \overline{\text{id}} \cdot \overline{\text{e}} \) and also the argument of the stuck case expression \( \overline{\text{id}} \cdot \overline{\text{e}} \) are needed to produce the strong normal form. Thus evaluation must continue with these redexes. That these redexes are indeed selected and, moreover, which one is selected first, depends on an appropriate notion of evaluation context. Our strategy will include an evaluation context \( C \) of the form \( (\text{case } (x y)(\overline{\text{y}} \cdot \overline{\text{c}}) \text{ of } c \Rightarrow d)(\overline{\text{id}} \cdot \overline{\text{e}}) \) and hence the body of the explicit substitution will be reduced next. Notice that in order for the focus of computation to be placed in the body of an explicit substitution, its target \( y \) should be needed. In this particular case, it is because \( x \) is free but \( y \) is needed for computing the strong normal form. However, in a term such \( \lambda x. c(y)(\overline{\text{id}} \cdot \overline{\text{id}}) \), the \( \beta \)-redex \( \overline{\text{id}} \cdot \overline{\text{id}} \) is not needed for the strong normal form and hence will not be selected by the strategy.

The remaining computation steps leading to the strong normal form are depicted below:
\[ (\text{case } (\lambda y. x y)(\overline{\text{id}} \cdot \overline{\text{id}}) \text{ of } c \Rightarrow d)(\overline{\text{id}} \cdot \overline{\text{e}}) \]
\[ \rightarrow_{sh} (\text{case } (x y)(\overline{\text{y}} \cdot \overline{\text{id}} \cdot \overline{\text{id}}) \text{ of } c \Rightarrow d)(\overline{\text{id}} \cdot \overline{\text{e}}) \]
\[ \rightarrow_{sh} (\text{case } (x y)(\overline{\text{y}} \cdot \overline{\text{id}} \cdot \overline{\text{z}} \cdot \overline{\text{y}})(\overline{\text{id}} \cdot \overline{\text{id}} \cdot \overline{\text{id}}) \text{ of } c \Rightarrow d)(\overline{\text{id}} \cdot \overline{\text{e}}) \]
\[ \rightarrow_{sh} (\text{case } (x y)(\overline{\text{y}} \cdot \overline{\text{id}} \cdot \overline{\text{z}})(\overline{\text{id}} \cdot \overline{\text{z}} \cdot \overline{\text{y}}) \cdot (\overline{\text{id}} \cdot \overline{\text{id}} \cdot \overline{\text{id}})(\overline{\text{z}} \cdot \overline{\text{z}}) \cdot (\overline{\text{id}} \cdot \overline{\text{z}} \cdot \overline{\text{e}})) \]
\[ \rightarrow_{sh} (\text{case } (x y)(\overline{\text{y}} \cdot \overline{\text{id}} \cdot \overline{\text{z}} \cdot \overline{\text{z}})(\overline{\text{id}} \cdot \overline{\text{id}} \cdot \overline{\text{id}})(\overline{\text{z}} \cdot \overline{\text{z}} \cdot \overline{\text{e}})) \]

Note that in the fourth step (indicated with an asterisk), \( y \) has been replaced by \( \overline{\text{id}} \). As in weak call-by-need, only answers shall be substituted for variables. Answers are abstractions under a possibly empty list of explicit substitutions or data structures possibly interspersed with explicit substitutions. Finally, crucial to defining the strong call-by-need strategy will be identifying variables and case expressions that will persist. The former are referred to as frozen variables and are free variables (or those that are bound under abstractions and branches of case expressions) that we know will never be substituted by an answer. The latter are referred to as error terms and are case expressions that we know will be stuck forever. An example of the former is \( x \cdot y \) in \( (x y)(\overline{\text{y}} \cdot \overline{\text{id}} \cdot \overline{\text{id}}) \); an example of the latter is case \( (x \cdot \overline{\text{id}})(\overline{\text{y}} \cdot \overline{\text{id}})(\overline{\text{z}} \cdot \overline{\text{y}})(\overline{\text{id}} \cdot \overline{\text{id}} \cdot \overline{\text{id}})(\overline{\text{z}} \cdot \overline{\text{z}} \cdot \overline{\text{e}})) \text{ of } c \Rightarrow d \) in \( (\text{case } (x \cdot \overline{\text{id}})(\overline{\text{y}} \cdot \overline{\text{id}})(\overline{\text{z}} \cdot \overline{\text{y}})(\overline{\text{id}} \cdot \overline{\text{id}} \cdot \overline{\text{id}})(\overline{\text{z}} \cdot \overline{\text{z}} \cdot \overline{\text{e}})) \text{ of } c \Rightarrow d)(\overline{\text{id}} \cdot \overline{\text{e}}) \).

Contribution. The main contributions of this paper are:
- A non-idempotent intersection type system for \( \lambda_e \) satisfying (2).
- A strong call-by-need strategy for \( \lambda_e \).
- A proof of completeness of the strategy.

Discussion. Although comparison with related work is developed in Sec. 7, we would like to comment on [9], the most closely related work (co-authored by two of the present authors). The work in [9] proposes a strong call-by-need strategy for the lambda calculus without matching and fixed point. It should perhaps be
mentioned that standard encodings of inductive types in the un-
typed lambda calculus such as the Church or Scott encodings do not
address the above mentioned problems. The culprit is the absence
of a high-level construct such as "case" which makes the notion of
"blocked case" not obvious in terms of the underlying encoding.
Eg., consider the standard Church encoding of a constant $c_1$ of arity
$1$:

$$\lambda x_1 \ldots x_n (l) c_1 \ldots c_n c_1 (x_1 \bar{c}) \ldots (x_n \bar{c})$$

How would the blocked case expression case $c$ of $(d \Rightarrow d)$ be
encoded? Resorting to the iterators of Church encodings, we would have:
$(\lambda c. d)? (\lambda c. d)$, where the question mark is the missing
branch. Consider also case $x$ of $(e \Rightarrow d; d \Rightarrow \Omega)$ with $x : [c]$. This would be encoded as $x (\lambda c. d) (\lambda c. d)$. The non-idempotent
intersection type system of $[9]$ does not account for $\Omega$.

**Definition 2.2.** The Extended Lambda Calculus $\Lambda_e$ is given by
the following reduction rules over $\Lambda_e$, closed by arbitrary contexts.
We write $\rightarrow_e$ for the resulting reduction relation.

$$\frac{}{(\lambda x.t)s \rightarrow_{e} \bar{t}}$$

$$\frac{x : f \Rightarrow t}{f(x)t \rightarrow_{e} f(x)t}$$

$$\frac{}{\text{case } c_i \text{ of } (c_i \bar{x}_i : s_i)_{i \in I} \rightarrow_{e} \text{case } s_j \bar{x}_j : i \text{ if } j \in I \text{ and } |I| = |\bar{x}_j|}$$

Capture-avoiding substitution of a variable $x$ by a term $s$ in a
term $t$ is written $t[x := s]$. Similarly, the simultaneous capture-
avoiding substitution of a list of variables $\bar{x}$ by a list of terms $\bar{s}$ of
the same length in a term $t$ is written $t[\bar{x} := \bar{s}]$. A term $t$ matches
with a branch $\bar{c} \bar{x} \Rightarrow s$ if $t = \bar{c} \bar{x} \Rightarrow s$ with $|\bar{x}| = |\bar{s}|$. A term $t$ matches with a list of branches if it matches with at least one branch. Given our
syntactic formation condition on case-expressions, in $\Lambda_e$ terms in
fact match with at most one branch. Note that term reduction may
become blocked if the condition of a case does not match any branch
(and never will). The normal forms of $\Lambda_e$ may be characterized as
follows:

**Lemma 2.3 (Normal forms).** The normal forms of $\Lambda_e$ are charac-
terized by the grammar:

$$N ::= \lambda x. xN | \lambda x.cN | \lambda x. \text{case } N_0 \text{ of } (c_i \bar{x}_i : s_i)_{i \in I} N$$

where $N_0$ does not match with $(c_i \bar{x}_i : s_i)_{i \in I}$. Note that the lists $\bar{x}$ and $N$ may be empty.

**Remark 2.** Since we work in an untyped setting blocked terms such
as case $c$ of $d \Rightarrow e$ must be admitted. In the Calculus of Inductive
Constructions, case analysis must be exhaustive.

We conclude this section with some standard terminology on
rewrite systems. Given a notion of reduction $R$ over a set of terms, we
use the following rewriting concepts. A term $t$ is in $R$-normal form
($R$-nf) if there is no $s$ such that $t \rightarrow_R s$. We write $\eta_R$ for
any term in $R$-normal form. We write $\rightarrow_R$ for the reflexive and
transitive closure of any reduction relation $\rightarrow_R$. A term $t$ is weakly
$R$-normalising, if there exists $s$ in $R$-normal form s.t. $t \rightarrow_R s$.
$\eta_R(s)$ denotes the set of $R$-normal forms and $\WN(\eta_R)$ the set of
weakly $R$-normalising terms. We write $\WN(\eta_R)$ for the reflexive,
symmetric, transitive closure of $\rightarrow_R$. We say that $t$ is definable as
$s$ in $\rightarrow_R$ if $t \equiv_R s$ for $s \in \WN(\eta_R)$. Also, $t$ is definable in $\rightarrow_R$ if it
is definable as $s$, for some $s$, in $\WN(\eta_R)$. We use "$\equiv"$ for definitional
equality.

**Remark 3.** $t$ is definable in $\Lambda_e$ iff $t \in \WN(\eta_e)$. This follows from
confluence of $\WN(\eta_e)$.

### 2.1 A Theory of Sharing (Step 1)

**Definition 2.4.** The terms of the theory of sharing terms $\Lambda_{sh}$
are defined as follows:

$$\text{free and bound variables are defined as expected. In particular, } x \text{ is bound by a fixed point operator } f(x), \text{ and all the variables } x_1 \ldots x_n \Rightarrow t. \text{Terms are considered up to renaming of bound variables. A context is a term } C \text{ with a single free occurrence of a hole } \bar{c}, \text{ and the variable-capturing substitution of the hole } \bar{c} \text{ by a term } t \text{ is written } C[t]; C[t][t] \text{ has the additional requirement that no free variables in } t \text{ are bound in } C.$$
A term \( t[x|s] \) is called a closure, and \( [x|s] \) is called an explicit substitution. Terms without explicit substitutions are called pure terms. Closures are often written as \( \lambda t \; x \; b \; s \) in \( t \) in the literature (e.g. [8]). The notions of free and bound variables of extended terms are defined as usual, in particular, \( \text{fv}(t[x|s]) = (\text{fv}(t) \setminus \{x\}) \cup \text{fv}(s) \) and \( \text{bv}(t[x|s]) = \text{bv}(t) \cup \{x\} \cup \text{bv}(s) \).

Definition 2.5. A pure term \( t^\circ \) is obtained from any \( t \in \Lambda_{\text{sh}} \) by unsharing:

\[
egin{align*}
    x^\circ & := x & f_{\lambda x. t}^\circ & := f_{\lambda x. t^\circ} \\
    e^\circ & := e & \text{(case of } b^\circ) & := \text{case } t^\circ \text{ of } b^\circ \\
    (\lambda x. t)^\circ & := \lambda x. t^\circ & (t[s^\circ])^\circ & := t^\circ[x := s^\circ] \\
    (t s)^\circ & := t^\circ s^\circ & (\text{c} x) & := c x^\circ \\
\end{align*}
\]

(e.g. \((\text{case } x \Rightarrow z)(z|d|d))\) = \( x|d|d \) of \( c \Rightarrow d \).

Additional syntactic categories will be required for describing reduction in \( \Lambda_{\text{sh}} \). First of all, in call-by-need computation one cannot substitute arbitrary terms for variables, rather one substitutes values for variables. In our theory of sharing apart from abstractions as values we also have terms headed by constants as values. Also, values may be embraced by pending explicit substitutions. This leads to the definition of answers.

**Answers**

\[
\begin{align*}
    a & := L[\{\}]; \\
    \text{Values} & := \lambda x. t | A[e]; \\
    \text{Constant Context} & := \emptyset | L[\{\}]; \\
    \text{Substitution Context} & := \emptyset | L[\{\}].
\end{align*}
\]

A term of the form \( L[\{\}] \) is sometimes written \( tl \) and called an answer. An answer of the form \( (\lambda x. t)L \) is an abstraction answer and one of the form \( A[e]t \) is a constant answer. An example of the latter is \( (\lambda x[I|y|d]|y|y|x] \).

Second, reduction in \( \Lambda_{\text{sh}} \) will take place under arbitrary contexts. We define such a set of full contexts next:

**Full Context**

\[
\begin{align*}
    C & := \emptyset | L[\{\}]; \\
     & \mid \lambda x. C | C[t] | f_{\lambda x. C}(x) \\
     & \mid \text{case } C \text{ of } b \\
     & \mid \text{case } t \text{ of } (e_1 x_1 \Rightarrow s_1) \ldots \text{ (case of } C \mid x \Rightarrow t \text{ of } C \mid x \rightarrow C) \\
     & \mid C[x|s] \mid t[x|C] \\
\end{align*}
\]

Definition 2.6. The theory of sharing \( \Lambda_{\text{sh}} \) consists of the reduction rules over \( \Lambda_{\text{sh}} \) given below, closed by full contexts. We write \( \rightarrow_{\text{sh}} \) for the reduction relation.

\[
\begin{align*}
    (\lambda x. t)Ls & \rightarrow_{\text{sh}} t[x|s]L; \\
    C[x|s]|C[t]L & \rightarrow_{\text{sh}} C[x|s]|C[t]L; \\
    t[x|s] & \rightarrow_{\text{gc}} t, \text{ if } x \notin \text{fv}(t) \\
    f_{\lambda x. t}[x|s] & \rightarrow_{\text{fix}} f_{\lambda x. t}[x|s]; \\
    \text{case } A[e]t & \rightarrow_{\text{sh}} \text{case } s j[x]L; \\
\end{align*}
\]

The \( \text{sh} \) rule transforms an application of an abstraction (possibly under multiple explicit substitutions) to an argument, into the body of the abstraction \( t \) subject to a new explicit substitution \( [x|s] \). The \( 1sv \) rule substitutes a free occurrence of \( x \) with the value \( v \). Since variables in \( t \) might be bound in \( L \) and we do not wish to duplicate \( L \), the scope of the substitution context is adjusted. This rule is said to operate at a distance since the explicit substitution is not required to propagate to variables before it is executed [6]. It is closely related with the notion of linear head reduction [4]. Rule gc removes garbage.
We call $e$ We write stuck also, we write $\bar{T}$

type are ranged over by $M, N, P, \ldots$. The empty multiset is written $[]$, and $[\tau_1, \ldots, \tau_n]$ stands for the multiset containing each of the types $\tau_i$ with their corresponding multiplicities. Moreover, $M + N$ stands for the (additive) union of multisets. For instance $[a, b] + [b, c] = [a, b, b, c].$

**Definition 3.1.** The types of $T$ are defined as follows:

<table>
<thead>
<tr>
<th>Types</th>
<th>$\tau$ ::= $\bar{\alpha} : M \to \tau \mid D \mid E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datatypes</td>
<td>$D ::= \bar{c} \mid D \mid M$</td>
</tr>
<tr>
<td>Error types</td>
<td>$E ::= (G) \mid E \tau$</td>
</tr>
<tr>
<td>Branch types</td>
<td>$B ::= \bar{\tau} \to \bar{\tau}$</td>
</tr>
</tbody>
</table>

The type $\bar{\alpha}$ is a type variable, $M \to \tau$ is a function type, $D$ is a datatype and $E$ is an error type. A datatype is either a constant type $c$ or an applied datatype $D \cdot M.$ Formally, if $M_1, \ldots, M_n$ is the type of a constant applied to $n$ arguments, each of which has been assigned a multiset of types. PreError types are solely introduced for building error types; error types are used for typing case expressions which will eventually become stuck. A case is stuck if, intuitively, it can be decided that the condition cannot match any branch. An error type $(\varepsilon \tau (M_i \Rightarrow \sigma_i)_{i \in I} (\rho_1 \ldots \rho_k))$ is the type of a case expression:

1. whose condition has type $\varepsilon$ and branches type $\bar{\sigma}$ in $\tau$;
2. which is stuck;
3. which has been applied to arguments of type $\rho_1 \ldots \rho_k$; and
4. which is expecting arguments of type $\rho_{j+1} \ldots \rho_k$.

We call $\varepsilon$ an error type constructor.

Letters $\Gamma, \Delta, \Theta, \ldots$ range over typing contexts, which are functions mapping variables to multisets of types. $\Gamma(x)$ is the multiset associated to the variable $x.$ $\text{dom} \Gamma$ is the domain of $\Gamma,$ namely the set of $x$ s.t. $\Gamma(x) \neq \emptyset.$

Letters $\Sigma, \Upsilon, \ldots$ range over error logs, sets of error types. The sum of typing contexts $\Gamma + \Delta$ is defined as follows: $\text{dom}(\Gamma + \Delta) := \text{dom} \Gamma \cup \text{dom} \Delta$ and $(\Gamma + \Delta)(x) := \Gamma(x) + \Delta(x).$ The disjoint sum of typing contexts $\Gamma \oplus \Delta$ is defined as $\Gamma + \Delta$ provided $\text{dom} \Gamma \cap \text{dom} \Delta = \emptyset.$ We write $\Gamma, x : M$ for $\Gamma + \{x : M\}$ and $\Gamma, x :: M$ for $\Gamma \oplus \{x : M\}.$ Also, we write $\bar{x} : \bar{M}$ for $(\{x_i\}_{i \in I} : (M_i)_{i \in I}) := \Sigma_{i \in I} (x_i : M_i)$ and similarly for $\bar{x} :: \bar{M}.$

**Definition 3.2.** The typing system $T$ is defined by means of the typing rules of Fig. 2. These rules introduce four, mutually recursive, typing judgements:

1. **Typing** ($\Gamma, \Sigma \vdash t : \tau$)
   Term $t$ has type $\tau$ under context $\Gamma$ and error log $\Sigma.$
2. **Multi-typing** ($\Gamma, \Sigma \vdash t : M$)
   Term $t$ has the types in $M$ under context $\Gamma$ and error log $\Sigma.$

(3) **Application** ($\tau @ M \Rightarrow \sigma$)
A term of type $\tau$ may be applied to an argument that has all the types in $M,$ resulting in a term of type $\sigma.$

(4) **Matching** ($\tau (\bar{b}) \Gamma, \Sigma, \sigma$)
Type $\tau$ might be the condition of a case with branches $\bar{b},$ which will result in a term of type $\sigma,$ provided certain hypotheses $\Gamma$ and error logs $\Sigma,$ or else fail.

We write $\pi, \xi, \ldots$ for typing derivations and $\pi(\Gamma, \Sigma \vdash t : \tau)$ if $\pi$ is a typing derivation of the judgement $\Gamma, \Sigma \vdash t : \tau.$ We comment on the salient typing rules. The axioms are linear w.r.t. the typing context in that they require the typing context to be empty but for the type assigned to $x$ in $\text{tVar}$ and the typing context to be empty in $\text{tCons}.$ The error context, however, is said to be intuitionistic in that it may hold any number of error types. Rule $\text{tApp}$ caters for typing applications of terms of functional type, data structures and error terms, to arguments by means of the application judgement $\tau @ M \Rightarrow \sigma.$ Indeed, $\tau$ may be of the form $M \to \sigma$ (cf. $\text{tAppFun}$), or a datatype $D$ in which case $\sigma$ is $DM$ (cf. $\text{tAppData}$), or an error type $G\tau_1\ldots\tau_n$ in which case $M$ must be a singleton $[\tau_1]$ and $\sigma$ of $G\tau_1\ldots\tau_n$ (cf. $\text{tAppErr}$). The restriction to a singleton type in the last case is due to the fact that all one wants to do is enforce that the arguments of a stuck case be typable. Note also that typing contexts are multiplicative whereas error logs are additive. The $\text{tFix}$ splits its resources so that they are dealt out to be used for one unfolding ($\Gamma$) and the rest of the unfoldings ($\Delta$). The $\text{tCase}$ rule relies on the matching judgement $\tau \to \bar{b} \Gamma, \Sigma, \sigma.$ The latter checks whether the type of the condition $\tau$ matches the list of branches. A type $\tau$ matches with a branch $\bar{x} \tau \Rightarrow \bar{s}$ if $\tau \in M$ with $|\bar{M}| = |\bar{x}|.$ A type $\tau$ matches with a list of branches if it matches with at least one branch. Returning to our description of $\text{tCase},$ if $\tau$ matches with a branch, then that branch is typed (cf. $\text{tCMatch}$). However, if $\tau$ does not match any branch (cf. $\text{tCMismatch}$), then all branches have to be accounted for by the type system. Moreover, the type of the case expression will be an error type of the form $(\varepsilon \tau (M_i \Rightarrow \sigma_i)_{i \in I}) \bar{\rho},$ which is recorded in the error log. Note that $\bar{\rho} = \rho_1 \ldots \rho_k$ are the types of the arguments to which the stuck case expression will be allowed to be applied. Finally, $\text{tMulti}$ allows a term to be typed with a multiset type. In this rule, if $n = 0,$ then $\Sigma_{i=1}^n [\tau_i]$ denotes the empty multiset $\emptyset.$

**Remark 5.** $T$ does not ensure unique typing. For example, it is possible to assign many types the expression $\text{cons zero}$ $(\text{cons zero nil},$ namely $\text{cons} [\bar{\text{z}}])$ or $\text{cons} [\bar{\text{zero}}].$

### 3.1 An Example
Let $t$ be the term $\text{fix}(f, \lambda n . \text{case } n \Rightarrow \bar{z} : s \cdot s n \Rightarrow s n * f n)$ representing the factorial function. We exhibit type derivations for the judgements:

1. $\emptyset \vdash \emptyset \vdash t : [z] \Rightarrow s [z]$; and
2. $\emptyset \vdash \emptyset \vdash t : [s [\bar{z}], z] \Rightarrow s [\bar{z}].$

We use $\bar{b}$ to denote the branches $\bar{z} \Rightarrow s z \cdot s n \Rightarrow s n * f n.$ The derivation $\pi$ for the first judgement is in Fig. 3(a). Note the absence of $f$ in the typing context of the judgement $\emptyset \vdash \emptyset \vdash \lambda n . \text{case } n \Rightarrow \bar{z} : s z ; s n \Rightarrow s n * f n ; [z] \Rightarrow s [z].$

Since the type of $n$ is $[\bar{z}]$ the branch with the recursive call will not be used and hence is not typed. The missing subderivation of $\pi$
Figure 2 Typing rules for \( \mathcal{T} \)

\[
\begin{array}{ll}
\text{TVar} & n: [\tau]; \varnothing \vdash n: [\tau] \\
\text{Case} & z \langle \hat{b} \rangle \varnothing; \varnothing, s[z] \\
\text{App} & \varnothing; \varnothing \vdash t: [\tau] \\
\text{Abs} & n: [\tau]; \varnothing \vdash \lambda x. t: M \Rightarrow \tau \\
\text{Fix} & \varnothing; \varnothing \vdash \text{fix}(x.t): M \Rightarrow \tau \\
\text{ES} & \Gamma; \Sigma \vdash \text{fix}(x.t): \tau \\
\text{Multi} & \sum_{i=1}^{n} \Gamma_i; \Sigma \vdash t: \sum_{i=1}^{n} [\tau_i] \\
\text{AppFun} & M \Rightarrow \tau \vdash \lambda \Delta; \Sigma \vdash s: [\tau] \\
\text{AppData} & M \Rightarrow \tau \vdash \text{case}\ M \Rightarrow \tau \vdash s: [\tau] \\
\end{array}
\]

(called \( \pi_1 \), of the judgement \( z \langle \hat{b} \rangle \varnothing, s[z] \), is given in Fig 3(b). Since \( z \) matches with \( z \Rightarrow s[z] \), we only type this branch.

A derivation \( \varnothing \vdash t: [s[z], z] \Rightarrow s[z] \) is in Fig. 3(c). We use the following typing rule for the product:
We introduce an extended set of constraints below that determines

\[ \Gamma; \Sigma \vdash t : s^n z \quad \Delta; \Sigma \vdash s : s^m z \quad \text{TProd} \]

The derivation \( \xi_1 \) of \( s[z] \) (def \( \{ f : [s] \rightarrow s \} \)); \( \phi, s[z] \) is given in Fig. 3(d). The missing subderivation \( \xi_2 \) may be completed without trouble.

4 TOWARDS COMPLETENESS OF THE STRATEGY

We next outline the proof method [9] that we use to prove that the

strong call-by-need strategy (to be introduced in Sec. 5) is correctly behaved w.r.t. reduction in \( \lambda_e \).

- **Step 1 (Sec. 4.1).** Weakly normalising terms of \( \lambda_e \) are typable in the non-idempotent intersection type system \( T \).
- **Step 2 (Sec. 4.2).** Typable terms in \( T \) are weakly normalising in the theory of sharing \( \lambda_{sh} \).
- **Step 3 (Sec. 6).** Factorization of the reduction sequence in \( \lambda_{sh} \), obtained in Step 2 into an external part followed by an internal part, by means of a standardisation theorem. The former part corresponds to the strong call-by-need strategy and the latter shown to be superfluous. The end term of the external part is shown to be identical modulo unsharing (or unfolding of explicit substitutions) to the original normal form from Step 1.

A corollary is that the strong call-by-need strategy is complete w.r.t reduction in the Extended Lambda Calculus: if \( t \) reduces to a normal form \( s \) in \( \lambda_e \), then the strategy computes a normal form \( u \) such that \( s \) is the unsharing of \( u \).

4.1 Definable Terms in \( \lambda_e \) are Typable (Step 1)

This section addresses Step 1 of the diagram below, where \( t \in \lambda_e \):

\[ \text{Step 1} \quad t \in \text{WN}(\lambda_e) \quad \Rightarrow \quad t \in \text{Typable}(T) \quad \Rightarrow \quad t \in \text{WN}(\lambda_{sh}) \]

As discussed in the Introduction, we don’t want \( t \) to *just* be typable in \( T \) but to be typable with some additional constraints so that Step 2 holds too. For example, we mentioned that \( x \in \Omega \) is typable by setting \( x \) to have type \( [] \rightarrow a \), for \( a \) a type variable; however, the term is not normalising in \( \lambda_{sh} \). We must require that the typable judgements \( \Gamma; \Sigma \vdash t : \tau \) such that \( [] \notin N(\Gamma) \) and \( \tau \notin N(\tau) \) [12].

Here \( N(\Gamma) \) and \( P(\tau) \) refer to the usual notions of negative and positive occurrences of types in \( \tau \) (cf. Fig. 4). In the presence of constants and case expressions, this constraint does not suffice. We introduce an extended set of constraints below that determines what we call **good judgements** (cf. Def. 4.1). We revisit below some examples from the introduction to motivate them.

Consider the term case \( e \) of \( (d \Rightarrow d) \). \( \Omega \) is typable with, for example, type \( (e \Rightarrow (d \Rightarrow d)) \). Notice how the type of the blocked case includes occurrences of the types of arguments to which it applies (in this case the empty multiset type). This will allow us to extend the above mentioned constraints to blocked case expressions.

Consider now the term case \( x \) of \( (e \Rightarrow d) \cdot (e \Rightarrow \Omega) \). This term is typable with type \( d \), if \( x \in [e] \); however, it is not weakly normalising in \( \lambda_{sh} \). This motivates the new constraints \( e \notin P(\Gamma), e \notin N(\Sigma) \), and \( e \notin N(\tau) \) in Def. 4.1. In particular, in a term such as case \( x \) of \( (e \Rightarrow d) \cdot (e \Rightarrow \Omega) \) which is in normal form, we will type it by assigning \( x \) an appropriate error type.

Finally, note that in pure lambda terms all terms in weak head normal form have a variable at the head. Since the types of all variables are in the typing context \( \Gamma \), we can place restrictions on their type through \( \Gamma \). For example, in the above mentioned term \( x \in \Omega \) one may require that \( [] \notin \Gamma(x) \) to force the type system to account for \( \Omega \). However, now we may have terms in weak head normal form headed by blocked case expressions. In order to have access to their types so that we may place restrictions on them, we have to record them. This is the role played by the error logs and motivates the third, and final, item of our notion of good judgements, namely that all occurrences of error types be accounted for in the error log.

**Definition 4.1 (Good types and typing judgements).**

The set of positive (resp. negative) types occurring in \( \tau \), denoted \( P(\tau) \) (resp. \( N(\tau) \)), is defined in Fig. 4. A type \( \tau \) is **good** if \( e \notin P(\tau) \) and \( [] \notin N(\tau) \).

We say \( M \) is good if each \( \tau \in M \) is good. A **typing context** \( \Gamma \) is **good** if \( \Gamma = \Gamma_g \Gamma_x \) and \( \forall x \in \text{dom} \Gamma_g, \Gamma_g(x) \) is good and \( \forall x \in \text{dom} \Gamma_x, \Gamma_x(x) \) is an error type. A **typing judgement** \( \Gamma; \Sigma \vdash t : \tau \) is **good** if

\( \Gamma \) is good;
\( [] \notin \Pi(\Sigma) \) and \( \tau \notin P(\tau) \);
\( e \notin N(\Sigma) \) and \( \tau \notin N(\tau) \); and

covered \( \xi_1(\Gamma) \) and covered \( \xi_2(\tau) \).

The proof of Step 1, namely that terms definable in \( \lambda_e \) terms are typable (Thm. 4.4), consists of two steps. First we show that \( \rightarrow_e \)-normal forms are typable with good typing judgements. These typing judgements \( \Gamma; \Sigma \vdash t : \tau \), for \( t \in \text{NF}(\rightarrow_e) \), are such that constants do not occur negatively in \( \tau \) nor in \( \Sigma \) nor positively in \( \Gamma \).

However, constants may occur positively in \( \tau \) such as when typing the normal form \( c \) and also negatively in \( \Gamma \) such as when typing the normal form \( x \in [\{c\} \rightarrow \alpha] \).

**Lemma 4.2 (Normal forms are typable).** Let \( t \in \text{NF}(\rightarrow_e) \). Then there exists a context \( \Gamma \), an error context \( \Sigma \), and a type \( \tau \) such that \( \pi(\Gamma; \Sigma \vdash t : \tau) \), and \( \Gamma; \Sigma \vdash t : \tau \) is good. Moreover, if \( t \) is of the form:

- \( x \in \text{N} \), then \( \tau \) is a type variable; and
- \( \text{(case } N_0 \text{ of } (c_i \in \Gamma) \Rightarrow (N_i)_{i \in I} \text{N}, \text{ where } N_0 \text{ does not match } (c_i \in \Gamma) \Rightarrow (N_i)_{i \in I} \text{N}, \text{ then } \tau = (e \tau \text{(M1 } \Rightarrow \alpha_1 \text{)} \cdot r_1 \cdot \ldots \cdot r_k) \text{ with } k = |\text{N}|. \)

The second step consists of showing subject expansion for \( \rightarrow_e \) (i.e. \( t \rightarrow_e s \) and \( \Gamma; \Sigma \vdash s : \tau \) imply \( \Gamma; \Sigma \vdash t : \tau \)).

**Lemma 4.3 (\( \rightarrow_e \)-expansion).** Let \( t \rightarrow_e s \). If \( \Gamma; \Sigma \vdash s : \tau \) then \( \Gamma; \Sigma \vdash t : \tau \).

**Theorem 4.4 (Step 1).** Suppose \( t \) is definable in \( \lambda_e \). Then there exists a context \( \Gamma \), an error context \( \Sigma \), a type \( \tau \) and a derivation \( \pi \) s.t. \( \pi(\Gamma; \Sigma \vdash t : \tau) \) with \( \Gamma; \Sigma \vdash t : \tau \) good.

4.2 Typable Terms are Definable in \( \lambda_{sh} \) (Step 2)

The idea behind Step 2 is to show that: 1) redexes in a term \( t \) that are accounted for by a typing derivation for \( t \), let’s call them **typed-redexes**, are finite in number and that that number can only decrease by reducing them; and 2) terms that are in such typed redex-normal form and that are typed with good typing judgements are also in normal form with respect to \( \lambda_{sh} \) (i.e. are in \( \text{NF}(\rightarrow_{sh}) \)). That a redex is accounted for in a typing derivation \( \pi \) is expressed as the redex occurring at a **typed occurrence** in \( \pi \). For example, in a term such as
Figure 4 Positive and negative types

\[ \begin{align*}
\mathcal{P}(a) & := \{a\} \\
\mathcal{P}(M \rightarrow \tau) & := \mathcal{N}(M) \cup \mathcal{P}(\tau) \cup \{M \rightarrow \tau\} \\
\mathcal{P}(c) & := \{c\} \\
\mathcal{P}(D \ M) & := \mathcal{P}(D) \cup \mathcal{P}(M) \cup \{D \ M\} \\
\mathcal{P}(\tau) & := \mathcal{P}(E) \cup \mathcal{P}(\tau) \cup \{E \tau\} \\
\mathcal{P}(G) & := \mathcal{P}(G) \cup \mathcal{P}(\tau) \cup \{G \tau\} \\
\mathcal{P}(\tau B) & := \mathcal{P}(\tau) \cup \mathcal{P}(\tau B) \cup \{\tau B\} \\
\mathcal{P}(M_1 \ldots M_n \Rightarrow \tau) & := \{\bigcup \mathcal{P}(\tau) \} \cup \{M \Rightarrow \tau\} \\
\mathcal{P}(M) & := \{N(\tau) \} \cup \mathcal{P}(\tau) \cup \{M\} \\
\mathcal{P}(\Gamma) & := \bigcup \mathcal{P}(\Gamma(x)) \\
\end{align*} \]

\[\begin{align*}
\mathcal{N}(a) & := \emptyset \\
\mathcal{N}(M \rightarrow \tau) & := \mathcal{P}(M) \cup N(\tau) \\
\mathcal{N}(c) & := \emptyset \\
\mathcal{N}(D \ M) & := \mathcal{N}(D) \cup \mathcal{N}(M) \\
\mathcal{N}(\tau) & := \mathcal{N}(E) \cup \mathcal{N}(\tau) \\
\mathcal{N}(G) & := \mathcal{N}(G) \\
\mathcal{N}(\tau B) & := \mathcal{N}(\tau) \cup \mathcal{N}(\tau B) \\
\mathcal{N}(M_1 \ldots M_n \Rightarrow \tau) & := \bigcup \mathcal{N}(\tau) \cup \{M \Rightarrow \tau\} \\
\mathcal{N}(M) & := \bigcup \mathcal{N}(\tau) \\
\mathcal{N}(\Gamma) & := \bigcup \mathcal{N}(\Gamma(x)) \\
\end{align*}\]

\[\begin{align*}
\text{covered}_2(\alpha) & := \text{true} \\
\text{covered}_1(\alpha) & := \text{covered}_2(\alpha) \land \text{covered}_2(\alpha) \\
\text{covered}_2(D) & := \text{covered}_2(D) \land \text{covered}_2(D) \\
\text{covered}_2(E) & := \text{covered}_2(E) \land E \in \Sigma \\
\text{covered}_1(D) & := \text{true} \\
\text{covered}_2(D) \land \text{covered}_2(D) & := \text{true} \\
\end{align*}\]

Theorem 4.5 (Step 2). If \( \pi(\Gamma; \Sigma \vdash t : \tau) \) and \( \Gamma; \Sigma \vdash t : \tau \) is good, then \( t \) is definable in \( \lambda_{sh} \).

Assembling Step 1 and Step 2 we obtain:

Theorem 4.6 (Soundness of \( \lambda_{sh} \) w.r.t. \( \lambda_e \)). Let \( t \) be a term in \( \lambda_e \). If \( t \in \mathcal{W}(\lambda_{sh}) \), then \( t \in \mathcal{W}(\lambda_{sh}) \). More precisely, if \( t \rightarrow_{e} n_e \), where \( n_e \in \lambda_{sh} \), is a \( \rightarrow_{e} \)-normal form, then \( t \rightarrow_{sh} n_{sh} \), where \( n_{sh} \) is a \( \rightarrow_{sh} \)-normal form. Moreover, \( n_{sh} = n_e \).

Proof. Let \( t \rightarrow_{e} n_e \), where \( n_e \in \rightarrow_{e} \)-nf. Then \( \pi(\Gamma; \Sigma \vdash t : \tau) \) and \( \Gamma; \Sigma \vdash t : \tau \) is good by Thm. 4.4. But then \( t \) is weakly \( \rightarrow_{sh} \)-normalising by Thm. 4.5, so that \( t \rightarrow_{sh} n_{sh} \), where \( n_{sh} \) is in \( \lambda_{sh} \)-nf. By Lem. 2.8(1) \( t \rightarrow_{sh} n_{sh} \) and by Lem. 2.8(2) \( n_{sh} \in \mathcal{W}(\lambda_{sh}) \). Since \( t \rightarrow_{e} n_e \) and \( t \rightarrow_{h} n^h_{sh} \), then we conclude \( n_{sh} = n_e \) because \( \rightarrow_{e} \equiv \) Church-Rosser.

5 THE STRONG CALL-BY-NEED STRATEGY

The strong call-by-need strategy \( \rightarrow_{sh} \) is a binary relation over terms in \( \lambda_{sh} \), and is parameterized over a set \( \theta \) of variables called frozen variables. It is defined by means of reduction rules similar to those given for the theory of sharing (Def. 2.6) only that the garbage collection rule is absent and reduction is restricted to a subset of the set of full contexts called evaluation contexts. We next describe evaluation contexts. Note that although they rely on a given set of normal forms, for expository purposes, we first describe the evaluation contexts and then characterize its normal forms.

Definition 5.1. Evaluation context judgments are expressions of the form \( C \in \mathcal{C}_{\theta}^0 \) where \( C \) is a full context, \( \theta \) is a set of variables and \( h \) is a symbol called discriminator of the context. This symbol may be one of ‘\( \cdot \), ‘\( \lambda \) or any constant \( e, d, \ldots \) and will prove convenient to discriminate the head symbol in the context; evaluation context formation rules will place requirements on them. An evaluation context is a context \( C \) such that the evaluation context judgement \( C \in \mathcal{C}_{\theta}^0 \) is derivable using the rules in Fig. 5.

\( \text{Box} \) states that any redex at the root is needed (we may disregard \( \cdot \) for now). Rule \( \text{App-L} \) allows reduction to take place to the left of an application. We must make sure that \( C \) is not an abstraction. This is achieved by requiring that \( h \neq \lambda \) (cf. \( \text{EAM} \) and how all rules persist \( h \)). Rule \( \text{AppRSTRUCT} \) allows reduction to take place to the right of an application when it is an argument of a term \( t \) that is a structure normal form or an error normal form. The \( \cdot \) in \( \tau \in \mathcal{C}_{\theta} \) reflects that \( \tau \) is not headed by a constant and that \( C \) is not an abstraction. Rule \( \text{AppRCONS} \) is similar only that the discriminator is set to the head variable of \( \tau \) via \( h \)-cut and will be consulted when deciding if reduction can take place in the condition of a case (cf. \( \text{ECLASS} \)). This function is defined as: \( h \rightarrow \cdot \) (cf. \( \text{EAM} \) and \( \text{ECLASS} \)). The role of frozen variables is best explained in the setting of \( \text{ECLASS} \) and \( \text{ECLASS} \). In a term \( t \) such as \( x[x'y's] \), clearly \( y \) is not to be substituted for \( z \) since it is not an error. Thus, computation has to proceed in \( s \). However, if \( t \) is placed under an explicit substitution, then whether we should reduce \( \cdot \) depends on its context. For example, we do want to reduce it in \( x[x'y's]\{y'[z] \} \) but not in \( x[x'y's]\{y'[z] \} \) since \( z \). does not use \( z \). These two examples motivate \( \text{ECLASS} \) (a is a structure normal form) and \( \text{ECLASS} \) (\( \lambda \) is not a structure normal form nor an error term). Also note that in order for the focus of computation to be placed to the right of \( y \) in \( y \), we must know that \( y \) will never be substituted for, or else, that it is frozen. Rule \( \text{ECLASS} \) allows computation to take place in the body of an explicit substitution.

There is no rule for \( \text{fix}(x.t) \) since reduction must take place at the root in a term such as that. Regarding case expressions, in order for reduction to take place in the condition we must ensure that reduction at the root is not possible (cf. \( \text{ECLASS} \)). This is achieved by requiring that the discriminator either is not a constant listed in the branches \( h \notin \{\sigma_1 \ldots \sigma_f \} \) or that, if it is, then the number of expected arguments by the branch are not met (\( |C[y]| \neq |x_j| \)). The notation \( |C[y]| \) counts the number of arguments of the spine of the term \( C[y] \). It is defined as follows:

\[\text{boxed}\]
Figure 5: Evaluation Contexts

\[
\begin{array}{ll}
\therefore \in C_\varnothing & \text{eBox} \\
C \in C^h_\varnothing & h \neq \lambda & \text{eApp} \\
\lep \in C^h_\varnothing & \text{eAppL} \\
C \in C^h_\varnothing & \text{eSubsR} \\
t \in S_\varnothing \cup L_\varnothing & \text{eAppRStruct} \\
C \in C^h_\varnothing & \text{eAppRCons} \\
\end{array}
\]

\[
\begin{array}{ll}
C \in C^h_\varnothing & x \notin S_\varnothing \cup L_\varnothing \quad \text{ESubsLNonStruct} \\
C \in C^h_\varnothing & \text{ESubsLStruct} \\
C_1 \in C^h_\varnothing & \text{ESubsR} \\
C_1 \in C_\varnothing & \text{eLAM} \\
C \in C^h_\varnothing & \phi(x) \in \varnothing, \psi(x) \in C^h_\varnothing & \text{eCase1} \\
C \in C_\varnothing & \phi(x) \in \varnothing, \psi(x) \in C_\varnothing & \text{eCase2} \\
\end{array}
\]

We know that in fact $C[y]$ is a constant answer:

**Lemma 5.2 (Answer contexts are answers).** Suppose $C \in C^h_\varnothing$.

- If $h = c$, then, for any term $t$, there exist $h$ and $L$ s.t. $C[t] = \lambda[c]L$.
- If $h \neq \lambda$, then, for any term $t$, there exists a variable $x$, term $s$ and substitution context $L$ s.t. $C[t] = (\lambda x.s)\Lambda$. Moreover, $C$ is either of the form $(\lambda x.C)\Lambda$ or $(\lambda x.t)\Lambda[y]C\Lambda[t_2]$.

For reduction to proceed in a branch $j$ (cf. eCase2), the condition must be in normal form, each branch $i$ with $i \neq j$ must be in normal form and the condition must not enable any branch ($t \neq (c_i x_i \Rightarrow s_i)_{i \neq j}$). Note that the bound variables in branch $j$ are added to the set of frozen variables. We now define the strategy itself.

**Definition 5.3.** The $\rightarrow^\varnothing_\sh$ strategy is defined by the following rules.

\[
\begin{align*}
C[(\lambda x.t)\Lambda] & \rightarrow^{\varnothing_\sh} C[(\lambda x.t)\Lambda] & \text{(DB)} \\
C[C(x)] & \rightarrow^{\varnothing_\sh} C[C(x)] & \text{(1sv)} \\
C[C(x)] & \rightarrow^{\varnothing_\sh} C[C(x)] & \text{(case)}
\end{align*}
\]

The discriminator $h$ in the conditions of all rules is existentially quantified. The condition $C_1[C_2[x]x[x]\Lambda]\in C^h_\varnothing$ in the definition of the 1sv-redex carries over from [9]. It avoids $1sv$-reducing $(\lambda x.g)(\bar{y})$ in $(\lambda x.g)(\bar{y})\Lambda$ so that the outermost $\bar{b}$-reduction step takes precedence instead. The condition also avoids $1sv$-reduce $x\Lambda y$ on the variable $x$, so that the $1sv$-reduction step on the variable $x$ takes precedence over it. The following result states that the strategy is deterministic:

**Lemma 5.4 (Determinism).** We say $r$ is an anchor in $C[r]$, if it is a $\bar{b}$-redex, a $f\bar{x}$-redex, a case-redex or a variable bound to an answer. If $C_1[r_1] = C_2[r_2]$, where $C_1,C_2 \in C^h_\varnothing$ and $r_1,r_2$ are anchors, then $C_1 = C_2$ and $r_1 = r_2$.

5.1 Normal Forms of the Strategy

We present an inductive characterization of the normal forms of $\rightarrow^\varnothing_\sh$. Since reduction in $\rightarrow^\varnothing_\sh$ is parameterized over a set of frozen variables $\varnothing$, the normal forms too will be parameterized by this set. The set of normal forms over $\varnothing (N_\varnothing)$ is comprised of the constant normal forms over $\varnothing (K_\varnothing)$, the structure normal forms over $\varnothing (S_\varnothing)$, the error normal forms over $\varnothing (E_\varnothing)$ and the lambda normal forms over $\varnothing (L_\varnothing)$. They are defined in Fig. 6 and are similar to the characterization of the $\rightarrow^\varnothing_\sh$-normal forms (Fig. 1) except that: 1) the set of frozen variables is tracked, 2) rule $\text{nfSub}$ is refined into rules $\text{nfSubN}$ and $\text{nfSubG}$ and 3) a new rule $\text{nfSubG}$ is added due to the absence of gc in $\rightarrow^\varnothing_\sh$. In rules $\text{nfSubN}$ and $\text{nfSubG}$, the symbol $x$ represents either $S_\varnothing, E_\varnothing, L_\varnothing$ or $K_\varnothing$. Rule $\text{nfSubG}$ helps capture terms such as $z(g[x]x\Lambda)$. Note that $x \in \text{fv}(g[x]x\Lambda)$ but this term is in normal form for any $s$. However, $x$ is not really "reachable" from $z$, it is would in fact be erased if we had gc. The notion of a variable being "reachable" in this sense is defined as follows:

**Definition 5.5.** The set of reachable or, better still, non-garbage variables of a term $t$, denoted $\text{ngv}(t)$, are defined below, where $b$ stands for $(c_i x_i \Rightarrow s_i)_{i \neq j}$.

\[
\begin{align*}
\text{ngv}(x) & := \{x\} \\
\text{ngv}(\lambda x.t) & := \text{ngv}(t) \setminus \{x\} \\
\text{ngv}(ts) & := \text{ngv}(t) \cup \text{ngv}(s) \\
\text{ngv}(f(x)) & := \text{ngv}(t) \setminus \{x\} \\
\text{ngv}(c) & := \emptyset \\
\text{ngv}(\text{case } t \rightarrow b) & := \text{ngv}(t) \cup \bigcup_{i \neq j} \text{ngv}(s_i) \setminus x_i \\
\text{ngv}(t[x/s]) & := \text{ngv}(t) \setminus \{x\} \cup \text{ngv}(s) \text{ if } x \in \text{ngv}(t) \\
\text{ngv}(t[x/s]) & := \emptyset \text{ otherwise}
\end{align*}
\]

The next result below states that Fig. 6 indeed characterizes the normal forms of the strategy.

**Lemma 5.6.** $\text{NF}(\rightarrow^\varnothing_\sh) = N_\varnothing$. 

6 A STANDARDIZATION THEOREM FOR THE THEORY OF SHARING

This section addresses a standardization theorem for λsh. Suppose $t$ is definable in $\lambda_{sh}$ as $n_{sh} \in N$. Then there is a reduction sequence $t \rightarrow_{sh} n_{sh}$ (cf. figure below). Notice that reduction steps in this sequence can take place under any context and substitution can take place even though the target is not needed for computing the strong normal-form. The standardization theorem reorganizes the computation steps in the reduction sequence $t \rightarrow_{sh} n_{sh}$ so that it can be factored into two parts $t \rightarrow_{sh|gc} u \rightarrow_{sh|gc} n_{sh}$. The prefix $t \rightarrow_{sh|gc} u$ is reduction via the strong call-by-need strategy; the double headed arrow indicates multiple steps of the strategy. The suffix $u \rightarrow_{sh|gc} n_{sh}$ consists of reduction steps in the theory that are internal, hence not required for obtaining the strong-normal form. In fact, $n_{sh}$ and $u$ are shown to be identical via unsharing. Moreover, $u$ is actually a normal form of the strategy. In summary, and following [9], the standardization theorem is split into three parts depicted below:

- Part I (Postponing gc): All gc steps are postponed (Lem. 6.1).
- Part II (Factorization): The resulting prefix is factorized into an external part that contributes to the strong normal form and an internal part that does not (Lem. 6.2).
- Part III (Internal steps are negligible). Internal steps all take place inside garbage explicit substitutions (Lem. 6.3).

Part I. Part I is just Lem. 6.1 below. A strict $\rightarrow_{sh}$-reduction step, denoted $\rightarrow_{sh|gc}$, is a $\rightarrow_{sh}$-reduction step without using the $\rightarrow_{gc}$-rule.

**Lemma 6.1 (Postponement of gc).** If $t \rightarrow_{sh} s$, then there is a term $u$ s.t. $t \rightarrow_{sh|gc} u \rightarrow_{gc} s$.

Part II. Part II requires that we first define what an internal step is. A $\vartheta$-internal $\rightarrow_{sh}$ step ($\rightarrow_{sh}^{\vartheta}$) is a $\rightarrow_{sh}$-step that is not a $\rightarrow_{sh}^{\vartheta}$-step, i.e., not a $\vartheta$-step in the strategy. External steps are steps in the strategy, that is, $\rightarrow_{sh}^{\vartheta}$-steps.

**Lemma 6.2 (Factorization of Strict Steps).** Let $fv(t) \subseteq \vartheta$. If $t \rightarrow_{sh|gc} u$, then there is a term $s$ such that $t \rightarrow_{sh|gc} u \rightarrow_{sh} s$.

Part III. As mentioned, if the $\rightarrow_{sh}$ reduction sequence reaches a $\rightarrow_{sh}$-normal form, then all the internal steps factored out by Lem. 6.2 can be erased by gc steps.

**Lemma 6.3 (Normal Forms Modulo Internal and gc Steps).** Let $\vartheta$, $t$ be such that $fv(t) \subseteq \vartheta$.

1. If $t \rightarrow_{gc} n_{\vartheta}$ with $n_{\vartheta} \in NF(\rightarrow_{sh}^{\vartheta})$ then $t \rightarrow_{NF}(\rightarrow_{sh}^{\vartheta})$.
2. If $t \rightarrow_{sh}^{\vartheta} n_{\vartheta}$ with $n_{\vartheta} \in NF(\rightarrow_{sh}^{\vartheta})$ then $t \rightarrow_{NF}(\rightarrow_{sh}^{\vartheta})$ and there is $u$ such that $t \rightarrow_{gc} u$ and $n_{\vartheta} \rightarrow_{gc} u$.

All three parts can now be assembled to complete the argument outlined in the Introduction.

**Theorem 6.4 (Standardization for $\rightarrow_{sh}$).** Let $\vartheta$. $t$ be such that $fv(t) \subseteq \vartheta$. If $t \rightarrow_{sh} n_{\vartheta}$, then $n_{\vartheta} \in NF(\rightarrow_{sh}^{\vartheta})$ and $n_{\vartheta} \rightarrow_{gc} n_{\vartheta}$.

**Corollary 6.5 (Completeness of $\rightarrow_{sh}^{\vartheta}$).** Let $\vartheta$. $t$ be such that $fv(t) \subseteq \vartheta$. If $t \rightarrow_{e} n_{\vartheta}$, then $n_{\vartheta} \in NF(\rightarrow_{e})$, then $n_{\vartheta} \rightarrow_{sh}^{\vartheta} n_{\vartheta}$.

7 RELATED WORK AND CONCLUSIONS

**Related Work.** Call-by-need for weak reduction was introduced in the 70s [22, 27]. Relating call-by-need strategies with call-by-need theories has been pioneered in [7, 13, 25]. Big-step semantics...
for call-by-need was studied in [24]. Completteness of call-by-need through intersection types was first studied in [23], although the result itself was proved by other means before that [7]. A recent survey on non-idempotent intersection types and its applications in the study of the lambda calculus may be found here [12]. The calculi with explicit substitutions at a distance used here is called the Linear Substitution Calculus and was inspired from [26] and further developed in [6]. The use of this tool to study abstract machines for weak call-by-need reduction appears here [1]. It is also used in [3], to provide a detailed analysis of the cost of adding pattern matching to β-reduction, although open terms are not considered.

Regarding strong reduction, as already mentioned in the introduction, [21] proposed an implementation of strong call-by-value, by iterating the standard call-by-value strategy on open terms (terms with variables). In [11], it is noted that the implementation of [21] requires modifying the OCAML abstract machine so they propose a native OCAML implementation where the tags that distinguish functions from accumulators are coded directly in OCAML itself. [15, 16] defined abstract machines for reduction to strong normal form. Other abstract machines for strong reduction have been studied too: [17–19]. [5] explore open call-by-value and [2] study a (call-by-name) machine based on the linear substitution calculus for reduction to strong normal form. None of these mentioned works address however strong call-by-need except for [9]. The latter proves similar results to this work but for β-reduction only. While developing this work we learned of [10]. In his PhD thesis, Bernadet proposes a non-idempotent intersection type system for a similar calculus that includes fixed-points and case expressions. The aim however is to characterize a subset of strongly normalising terms. Thus, for example, the standard fixed-point combinators used here cannot be typed; a modified combinator is adopted. Since there is no notion of call-by-need reduction strategy, ideas related to good or covered types, as presented here, are not developed either.

**Conclusions.** The recent formulation of a strong call-by-need strategy [9] was argued to provide a foundation for checking conversion in proof assistants. This work emerged out of the realization that the restriction to β-reduction of [9], and hence lack of treatment of inductive types and fixed point operators, left a gap to be filled. We have introduced a strong call-by-need strategy that is proved to be complete with respect to the Extended Lambda Calculus of Grégoire and Leroy [21] that includes the aforementioned constructs. A key obstacle has been devising a non-idempotent intersection type system that could connect reduction in the Extended Lambda Calculus with reduction in the theory of sharing, the latter is also introduced in this paper. This system is able to deal with case expressions that can block on open terms or non-exhaustive branches and also that can collect arguments. The presence of the fixed-point combinator has not provided any substantial obstacles.

In order to base an implementation of conversion in a proof assistant on our strategy, one should be able to iterate a restriction of it, to weak head normal form, as described in [14]. This has the benefit of failing early when types are not equivalent. Another line of work is to implement a compiled version of the strategy, as developed in [21]. Finally, big-step semantics and abstract machines that implement our strategy are yet to be developed.

**REFERENCES**


