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 (λ_e), as introduced by Gregoire and Leroy. The λ_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 λ_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:
Pablo Barenbaum, Eduardo Bonelli, and Kareem Mohamed. 2018. Pattern Matching and Fixed Points: Resource Types and Strong Call-By-Need: Extended Abstract. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany,* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.1145/3236950.3236972

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, $n \geq 0$:

$$pow := \lambda n.\lambda m.\text{if} n = 0 \text{ then } 1 \text{ else } m * pow\ (n-1)\ m$$

If we know the value of n to be 2, then we can produce a more efficient version of $pow\ 2$ as follows:

$$\begin{aligned}
pow\ 2 &\longrightarrow \lambda m.\text{if } 2 = 0 \text{ then } 1 \text{ else } m * (pow\ (2-1)\ m) \\
&\longrightarrow \lambda m.m * (pow\ (2-1)\ m) \\
&\longrightarrow \lambda m.m * (\text{if } 1 = 0 \text{ then } 1 \text{ else } m * (pow\ (1-1)\ m)) \\
&\longrightarrow \lambda m.m * (m * (pow\ (1-1)\ m)) \\
&\longrightarrow \lambda m.m * (m * (\text{if } 0 = 0 \text{ then } 1 \text{ else } m * (pow\ (0-1)\ m))) \\
&\longrightarrow \lambda m.m * (m * 1) \\
&\longrightarrow \lambda m.m * m
\end{aligned}$$

Notice that all the reduction steps depicted above, take place under the lambda abstraction λm. 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*:

$$\frac{\Gamma \vdash t : \tau \quad \tau \equiv \sigma}{\Gamma \vdash t : \sigma}$$

Checking that types τ and σ 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.

*Produces the permission block, and copyright information

PPDP '18, September 3–5, 2018, Frankfurt am Main, Germany

ACM ISBN 978-1-4503-6441-6/18/09…$15.00
https://doi.org/10.1145/3236950.3236972

The Extended Lambda Calculus. The Extended Lambda Calculus [21] (referred in *op.cit.* as the "type-erased lambda calculus"), denoted λ_{e}, extends the lambda calculus with constants, pattern matching and fixed-points. Here is an example of a term in λ_{e} that computes the length of a list encoded with constants **nil** and **cons** (see Sec. 2 for a detailed definition of λ_{e}):

$$\mathtt{fix}\,(l.\ \lambda xs.\ \mathtt{case}\ xs\ \mathtt{of}\ (\mathbf{nil} \Rightarrow \mathbf{zero}) \cdot (\mathbf{cons}\ hd\ tl \Rightarrow \mathbf{succ}\,(l\ tl)))$$

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 λ_{e} in order to apply it to check type-conversion. They propose a notion of strong reduction for λ_{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 λ_{e} term, where id abbreviates the identity term $\lambda z.z$:

$$\mathtt{case}\ \mathbf{c}\,(\mathtt{id}\ \mathtt{id})\ \mathtt{of}\ \mathbf{c}\,x \Rightarrow \mathbf{d} \tag{1}$$

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

Adding Sharing to Strong Reduction. This paper proposes a notion of strong reduction for λ_{e} that only reduces redexes that are needed in order to obtain the (strong) normal form. *e.g.* our strategy will not reduce the id 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 λ_{e} with an additional syntactic construct to hold such suspended arguments: *explicit substitutions*. The resulting theory of sharing (λ_{sh}) replaces the usual β rule in the lambda calculus $(\lambda x.t)\,s \to_\beta t\{x := s\}$ with $(\lambda x.t)\,s \to_{\mathsf{dB}} t[x\backslash s]$. The term $t[x\backslash s]$ is also often written $\mathtt{let}\ x = s\ \mathtt{in}\ t$. 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 $(\lambda x.x)[x\backslash y]\,z$, a slightly more general formulation of dB is adopted, namely $(\lambda x.t)\mathtt{L}\ s \to_{\mathsf{dB}} t[x\backslash s]\mathtt{L}$. 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: $\mathtt{C}[\![x]\!][x\backslash v\mathtt{L}] \to_{\mathtt{lsv}} \mathtt{C}[v][x\backslash v]\mathtt{L}$. Note how this rule makes use of a context C and the notation $\mathtt{C}[\![x]\!]$ to mean that there is a free occurrence of x. For example, $(x\,x)[x\backslash\lambda y.y] \to_{\mathtt{lsv}} ((\lambda y.y)\,x)[x\backslash\lambda y.y]$ and also $(x\,x)[x\backslash\lambda y.y] \to_{\mathtt{lsv}} (x\,(\lambda y.y))[x\backslash\lambda y.y]$. Of course, also $((\lambda y.z)\,x)[x\backslash\lambda y.y] \to_{\mathtt{lsv}} ((\lambda y.z)\,(\lambda y.y))[x\backslash\lambda y.y]$, 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 $\to_{\mathsf{sh}}$, 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 λ_{e} that has a normal form *also* has a normal form in λ_{e} *with sharing* (λ_{sh}). That is, that the restricted notion of replacement of results is general enough to capture all normalising derivations in λ_{e}.

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

The notation $\mathsf{WN}(\lambda_{\mathsf{e}})$ denotes the set of λ_{e}-terms that have a normal form via λ_{e} and similarly for $\mathsf{WN}(\lambda_{\mathsf{sh}})$. 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, redex tracking and dags with boxes. It was recently noticed [23] that by devising an appropriate *non-idempotent intersection type system* $\mathcal{T}$ for λ_{sh}, one could achieve this as follows:

$$t \in \mathsf{WN}(\lambda_{\mathsf{e}}) \overbrace{\Rightarrow}^{\textit{Step 1}} t \in \mathsf{Typable}(\mathcal{T}) \overbrace{\Rightarrow}^{\textit{Step 2}} t \in \mathsf{WN}(\lambda_{\mathsf{sh}}) \tag{2}$$

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, $y : [[\tau_1, \tau_2] \to \tau_3]$ means that y has to be used once and that the argument to y has to be typed twice, once with type τ_1 and once with τ_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 Ω is $(\lambda x.x\,x)\,(\lambda x.x\,x)$, is typable by setting x to have type $[] \to \alpha$, for α a type variable; the empty multiset of type [] allowing the typing of Ω to *not* be accounted for. However, the term is not normalising in λ_{sh} or any theory of sharing that allows β 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:

$$\mathtt{case}\ \mathbf{c}\ \mathtt{of}\ (\mathbf{c} \Rightarrow \mathbf{d}) \cdot (\mathbf{d} \Rightarrow \Omega)$$

It will evaluate to $\mathbf{d}$ and hence should be typable in $\mathcal{T}$ (*cf.* Step 1). Since Ω does not participate at all in computing $\mathbf{d}$, 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:

$$\mathsf{case}\ \mathbf{c}\ \mathsf{of}\ (\mathbf{d} \Rightarrow \mathbf{d}) \cdot (\mathbf{e} \Rightarrow \mathbf{e})$$

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:

$$\mathsf{case}\ \mathbf{c}\ \mathsf{of}\ (\mathbf{d} \Rightarrow \mathbf{d}) \cdot (\mathbf{e} \Rightarrow \Omega)$$

where, although matching is blocked, have not strong normal form in λ_{e} or λ_{sh}. Since blocked case expressions could be applied to arguments, further considerations are required. Consider the term:

$$(\mathsf{case}\ \mathbf{c}\ \mathsf{of}\ \mathbf{d} \Rightarrow \mathbf{d})\,\Omega$$

It does not have a normal form in λ_{e} or λ_{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 Ω, so that constraints on these types may be placed. In other words, we need to devise $\mathcal{T}$ such that it gives $\mathsf{case}\ \mathbf{c}\ \mathsf{of}\ (\mathbf{d} \Rightarrow \mathbf{d})\,\Omega$ a type that *includes* that of Ω. This would enable us to state conditions that do not allow this term to be typed but do allow a term such as $(\mathsf{case}\ \mathbf{c}\ \mathsf{of}\ \mathbf{d} \Rightarrow \mathbf{d})\,\mathbf{e}$ 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:

$$\mathsf{case}\ x\ \mathsf{of}\ (\mathbf{c} \Rightarrow \mathbf{d}) \cdot (\mathbf{e} \Rightarrow \Omega)$$

Although it does not have a normal form in λ_{sh}, it is typable with type $\mathbf{d}$, if $x : [\mathbf{c}]$. Notice, moreover, that the empty multiset 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 Ω is never accounted: since x is known to have type $[\mathbf{c}]$, only the $\mathbf{c} \Rightarrow \mathbf{d}$ 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 contant types such as $\mathbf{c}$ 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 $x\,\mathbf{c}$ 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 λ_{e} to be typable in $\mathcal{T}$. 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 $\rightarrow_{\mathsf{sh}}$ 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 $\rightarrowtail_{\mathsf{sh}}$, "sh" for sharing. Consider the term:

$$(\mathsf{case}\ (\lambda y.x\,y)(\mathsf{id}\ \mathsf{id})\ \mathsf{of}\ \mathbf{c} \Rightarrow \mathbf{d})\ (\mathsf{id}\,\mathbf{c})$$

It consists of a case expression applied to an argument. This case expression has a *condition* $(\lambda y.x\,y)(\mathsf{id}\ \mathsf{id})$, a *branch* $\mathbf{c} \Rightarrow \mathbf{d}$ with *pattern* $\mathbf{c}$ and *target* $\mathbf{d}$, and is applied to an argument $\mathsf{id}\,\mathbf{c}$. The first reduction step for this term is the same as for weak call-by-need, namely reducing the β-redex $(\lambda y.x\,y)(\mathsf{id}\ \mathsf{id})$ in the condition of the case. It must be reduced in order to determine which branch, if any, is to be selected. This β-redex is turned into $(x\,y)[y\backslash \mathsf{id}\ \mathsf{id}]$. The resulting term is:

$$(\mathsf{case}\ (x\,y)[y\backslash \mathsf{id}\ \mathsf{id}]\ \mathsf{of}\ \mathbf{c} \Rightarrow \mathbf{d})(\mathsf{id}\,\mathbf{c})$$

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 $\mathsf{id}\ \mathsf{id}$ and also the argument of the stuck case expression $\mathsf{id}\,\mathbf{c}$ 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 $\mathtt{C}$ of the form $(\mathsf{case}\ (x\,y)[y\backslash\Box]\ \mathsf{of}\ \mathbf{c} \Rightarrow \mathbf{d})(\mathsf{id}\,\mathbf{c})$ 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 $\lambda x.\mathbf{c}[y\backslash \mathsf{id}\ \mathsf{id}]$, the β-redex $\mathsf{id}\ \mathsf{id}$ 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.

$$\begin{array}{ll} & (\mathsf{case}\ (\lambda y.x\,y)(\mathsf{id}\ \mathsf{id})\ \mathsf{of}\ \mathbf{c} \Rightarrow \mathbf{d})(\mathsf{id}\,\mathbf{c}) \\ \rightarrowtail_{\mathsf{sh}} & (\mathsf{case}\ (x\,y)[y\backslash \mathsf{id}\ \mathsf{id}]\ \mathsf{of}\ \mathbf{c} \Rightarrow \mathbf{d})(\mathsf{id}\,\mathbf{c}) \\ \rightarrowtail_{\mathsf{sh}} & (\mathsf{case}\ (x\,y)[y\backslash z[z\backslash \mathsf{id}]]\ \mathsf{of}\ \mathbf{c} \Rightarrow \mathbf{d})(\mathsf{id}\,\mathbf{c}) \\ \rightarrowtail_{\mathsf{sh}} & (\mathsf{case}\ (x\,y)[y\backslash \mathsf{id}[z\backslash \mathsf{id}]]\ \mathsf{of}\ \mathbf{c} \Rightarrow \mathbf{d})(\mathsf{id}\,\mathbf{c}) \\ \rightarrowtail_{\mathsf{sh}} & (\mathsf{case}\ (x\ \mathsf{id})[y\backslash \mathsf{id}][z\backslash \mathsf{id}]\ \mathsf{of}\ \mathbf{c} \Rightarrow \mathbf{d})(\mathsf{id}\,\mathbf{c}) \qquad (*) \\ \rightarrowtail_{\mathsf{sh}} & (\mathsf{case}\ (x\ \mathsf{id})[y\backslash \mathsf{id}][z\backslash \mathsf{id}]\ \mathsf{of}\ \mathbf{c} \Rightarrow \mathbf{d})(z[z\backslash \mathbf{c}]) \\ \rightarrowtail_{\mathsf{sh}} & (\mathsf{case}\ (x\ \mathsf{id})[y\backslash \mathsf{id}][z\backslash \mathsf{id}]\ \mathsf{of}\ \mathbf{c} \Rightarrow \mathbf{d})(\mathbf{c}[z\backslash \mathbf{c}]) \end{array}$$

Note that in the fourth step (indicated with an asterisk), y has been replaced by 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 $x\,y$ in $(x\,y)[y\backslash \mathsf{id}\ \mathsf{id}]$; an example of the latter is $\mathsf{case}\ (x\ \mathsf{id})[y\backslash \mathsf{id}][z\backslash \mathsf{id}]\ \mathsf{of}\ \mathbf{c} \Rightarrow \mathbf{d}$ in $(\mathsf{case}\ (x\ \mathsf{id})[y\backslash \mathsf{id}][z\backslash \mathsf{id}]\ \mathsf{of}\ \mathbf{c} \Rightarrow \mathbf{d})(\mathsf{id}\,\mathbf{c})$.

Contribution. The main contributions of this paper are:

- A non-idempotent intersection type system for λ_{e} satisfying (2).
- A strong call-by-need strategy for λ_{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 untyped 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 $\mathbf{c}_i$ of arity $a(i)$:

$$\lambda x_1 \dots x_{a(i)}.\lambda c_1 \dots c_n.c_i\,(x_1\,\vec{c})\dots(x_n\,\vec{c})$$

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

Structure of the paper. We revisit the Extended Lambda Calculus λ_{e} in Sec. 2 and also introduce the theory of sharing for it λ_{sh} in the same section. Sec. 3 introduces the type system $\mathcal{T}$. Sec. 4 addresses Step 1 and 2 as described above. We present the strong call-by-need strategy in Sec. 5 and address the final completeness result (standardization theorem) in Sec. 6. Finally, we comment on related work and conclude, suggesting further avenues to pursue.

2 A THEORY OF SHARING FOR THE EXTENDED LAMBDA CALCULUS

We assume given a denumerable set of variables $x, y, z, \dots$ and constants $\mathbf{c}, \mathbf{c}', \mathbf{c}'', \dots$.

Definition 2.1. The **terms of the Extended Lambda Calculus** Λ_{e} are defined as follows:

Terms	$t, s, u, \dots$	::=	$x \mid \lambda x.t \mid t\,s \mid \mathbf{c} \mid \mathtt{case}\ t\ \mathtt{of}\ \bar{b} \mid \mathtt{fix}(x.t)$
Branch	b	::=	$\mathbf{c}\bar{x} \Rightarrow t$
Contexts	$\mathtt{C}$	::=	$\Box \mid \lambda x.\mathtt{C} \mid \mathtt{C}\,t \mid t\,\mathtt{C} \mid \mathtt{fix}(x.\mathtt{C}) \mid \mathtt{case}\ \mathtt{C}\ \mathtt{of}\ \bar{b}$
			$\mid \mathtt{case}\ t\ \mathtt{of}\ (\mathbf{c}_1\bar{x}_1 \Rightarrow s_1)\dots$
			$\dots(\mathbf{c}_i\bar{x}_i \Rightarrow \mathtt{C})\dots(\mathbf{c}_n\bar{x}_n \Rightarrow s_n)$

In addition to the standard terms of the lambda calculus, **variables**, **abstraction** and **application**, we have **constants**, **case expressions** and **fixed-point expressions**. In $\mathtt{case}\ t\ \mathtt{of}\ \bar{b}$ we say t is the **condition** of the case and $\bar{b}$ are its **branches**; $\bar{b}$ is shorthand for a (possibly empty) sequence of branches. If $I = \{1, 2, \dots, n\}$, we sometimes write $\mathtt{case}\ t\ \mathtt{of}\ (\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I}$ for case expressions. Branches are assumed to be syntactically restricted so that if $i \neq j$ then $(\mathbf{c}_i, |\bar{x}_i|) \neq (\mathbf{c}_j, |\bar{x}_j|)$, where $|\bar{x}_j|$ denotes the length of the sequence $\bar{x}_j$. Moreover, the list $\bar{x}_i$ of formal parameters in each branch is assumed to have no repeats. We write $\mathtt{fix}(x.t)$ for the standard fixed-point expression. We often write $\lambda\bar{x}.t$ for $\lambda x_1.\dots.\lambda x_n.t$ if $\bar{x}$ is the sequence of variables $x_1 \cdot \dots \cdot x_n$ and similarly $t\bar{s}$ stands for $ts_1 \dots s_n$ if $\bar{s} = s_1 \cdot \dots \cdot s_n$. **Free** and **bound** variables are defined as expected. In particular, x is bound by a fixed point operator $\mathtt{fix}(x.t)$, and all the variables $x_1, \dots, x_n$ are bound in a branch $\mathbf{c}x_1 \dots x_n \Rightarrow t$. Terms are considered up to renaming of bound variables. A **context** is a term $\mathtt{C}$ with a single free occurrence of a hole $\Box$, and the variable-capturing substitution of the hole $\Box$ by a term t is written $\mathtt{C}[t]$; $\mathtt{C}[\![t]\!]$ has the additional requirement that no free variables in t are bound in $\mathtt{C}$.

Remark 1. *In [21] a family of fixed-point operators fix_n, for n a positive integer, is used. The index n indicates the expected number of arguments and also the index of the argument that is used to guard recursion to avoid infinite unfoldings. The type system of the Calculus of Constructions guarantees that the recursive function is applied to strict subterms of the n-th argument. Although we use the more general fixed-point operator fix in our calculus similar ideas to "case" can be applied to fix_n which "blocks" if given less than n arguments.*

Definition 2.2. The **Extended Lambda Calculus** λ_{e} is given by the following reduction rules over Λ_{e}, closed by arbitrary contexts. We write $\to_{\mathsf{e}}$ for the resulting reduction relation.

$(\lambda x.t)s$	$\mapsto_{\mathsf{dB}}$	$t\{x := s\}$	(β)
$\mathtt{fix}(x.t)$	$\mapsto_{\mathtt{fix}}$	$t\{x := \mathtt{fix}(x.t)\}$	$(\mathtt{fix})$
$\mathtt{case}\ \mathbf{c}_j\bar{t}\ \mathtt{of}\ (\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I}$	$\mapsto_{\mathtt{case}}$	$s_j\{\bar{x}_j := \bar{t}\}$ if $j \in I$ and $\lvert\bar{t}\rvert = \lvert\bar{x}_j\rvert$	$(\mathtt{case})$

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 $\mathbf{c}\bar{x} \Rightarrow s$ if $t = \mathbf{c}\bar{s}$ with $|\bar{s}| = |\bar{x}|$. 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 λ_{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 λ_{e} may be characterized as follows:

Lemma 2.3 (Normal forms). *The normal forms of λ_{e} are characterized by the grammar:*

$$N \quad ::= \quad \lambda\bar{x}.x\bar{N} \mid \lambda\bar{x}.\mathbf{c}\bar{N} \mid \lambda\bar{x}.(\mathtt{case}\ N_0\ \mathtt{of}\ (\mathbf{c}_i\bar{x}_i \Rightarrow N_i)_{i\in I})\bar{N}$$

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

Remark 2. *Since we work in an untyped setting blocked terms such as $\mathtt{case}\ \mathbf{c}\ \mathtt{of}\ \mathbf{d} \Rightarrow \mathbf{e}$ 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 $\mathcal{R}$ over a set of terms, we use the following rewriting concepts. A term t is in **$\mathcal{R}$-normal form** ($\mathcal{R}$-nf) if there is no s such that $t \to_{\mathcal{R}} s$. We write $\mathsf{n}_{\mathcal{R}}$ for any term in $\mathcal{R}$-normal form. We write $\twoheadrightarrow_{\mathcal{R}}$ for the reflexive and transitive closure of any reduction relation $\to_{\mathcal{R}}$. A term t is **weakly $\mathcal{R}$-normalising**, if there exists s in $\mathcal{R}$-normal form s.t. $t \twoheadrightarrow_{\mathcal{R}} s$. $\mathsf{NF}(\to_{\mathcal{R}})$ denotes the set of $\mathcal{R}$-normal forms and $\mathsf{WN}(\to_{\mathcal{R}})$ the set of weakly $\mathcal{R}$-normalising terms. We write $\leftrightarrow^*_{\mathcal{R}}$ for the reflexive, symmetric, transitive closure of $\to_{\mathcal{R}}$. We say that t is **definable as** s in $\to_{\mathcal{R}}$, if $t \leftrightarrow^*_{\mathcal{R}} s$ for $s \in \mathsf{NF}(\mathcal{R})$. Also, t is **definable** in $\to_{\mathcal{R}}$ if it is definable as s, for some s, in $\to_{\mathcal{R}}$. We use ":=" for definitional equality.

Remark 3. *t is definable in λ_{e} iff $t \in \mathsf{WN}(\to_{\mathsf{e}})$. This follows from confluence of $\to_{\mathsf{e}}$.*

2.1 A Theory of Sharing (Step 1)

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

$$t, s, u, \dots \quad ::= \quad x \mid \lambda x.t \mid t\,s \mid \mathtt{fix}(x.t) \mid \mathbf{c} \mid \mathtt{case}\ t\ \mathtt{of}\ \bar{b} \mid t[x\backslash s]$$

A term $t[x\backslash s]$ is called a **closure**, and $[x\backslash s]$ is called an **explicit substitution**. Terms without explicit substitutions are called **pure terms**. Closures are often written as $\texttt{let}\ x\ \texttt{be}\ s\ \texttt{in}\ t$ in the literature (*e.g.* [8]). The notions of **free** and **bound variables** of extended terms are defined as usual, in particular, $\mathsf{fv}(t[x\backslash s]) = (\mathsf{fv}(t) \setminus \{x\}) \cup \mathsf{fv}(s)$ and $\mathsf{bv}(t[x\backslash s]) = \mathsf{bv}(t) \cup \{x\} \cup \mathsf{bv}(s)$.

Definition 2.5. A pure term $t^\diamond$ is obtained from any $t \in \Lambda_{\mathsf{sh}}$ by **unsharing**:

$$\begin{array}{rclcrcl} x^\diamond & := & x & & \mathtt{fix}(x.t)^\diamond & := & \mathtt{fix}(x.t^\diamond) \\ \mathbf{c}^\diamond & := & \mathbf{c} & & (\mathtt{case}\ t\ \mathtt{of}\ \bar{b})^\diamond & := & \mathtt{case}\ t^\diamond\ \mathtt{of}\ \bar{b^\diamond} \\ (\lambda x.t)^\diamond & := & \lambda x.t^\diamond & & (t[x\backslash s])^\diamond & := & t^\diamond\{x := s^\diamond\} \\ (t\,s)^\diamond & := & t^\diamond\, s^\diamond & & (\mathbf{c}\bar{x} \Rightarrow t)^\diamond & := & \mathbf{c}\bar{x} \Rightarrow t^\diamond \end{array}$$

e.g. $((\mathtt{case}\ z\ \mathtt{of}\ \mathbf{c} \Rightarrow z)[z\backslash \mathbf{d}\,\mathbf{d}])^\diamond = \mathtt{case}\ \mathbf{d}\,\mathbf{d}\ \mathtt{of}\ \mathbf{c} \Rightarrow \mathbf{d}\,\mathbf{d}$. Additional syntactic categories will be required for describing reduction in λ_{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	a	::=	$\mathsf{L}[v]$
Values	v	::=	$\lambda x.t \mid \mathsf{A}[\mathbf{c}]$
Constant Context	A	::=	$\Box \mid \mathsf{L}[\mathsf{A}]\,t$
Substitution Context	L	::=	$\Box \mid \mathsf{L}[x\backslash t]$

A term of the form $\mathsf{L}[v]$ is sometimes written $v\mathsf{L}$ and called an **answer**. An answer of the form $(\lambda x.t)\mathsf{L}$ is an **abstraction answer** and one of the form $\mathsf{A}[\mathbf{c}]\mathsf{L}$ is a **constant answer**. An example of the latter is $((\mathbf{c}\,x)[x\backslash y]\,\mathbf{d})[y\backslash s]$.

Second, reduction in λ_{sh} will take place under arbitrary contexts. We define such a set of full contexts next:

$$\begin{array}{rrcl} \textbf{Full Context} & \mathsf{C} & ::= & \Box \mid \lambda x.\mathsf{C} \mid \mathsf{C}\,t \mid t\,\mathsf{C} \mid \mathtt{fix}(x.\mathsf{C}) \\ & & \mid & \mathtt{case}\ \mathsf{C}\ \mathtt{of}\ \bar{b} \\ & & \mid & \mathtt{case}\ t\ \mathtt{of}\ (\mathbf{c}_1\bar{x}_1 \Rightarrow s_1)\ldots \\ & & & \ldots(\mathbf{c}_i\bar{x}_i \Rightarrow \mathsf{C})\ldots(\mathbf{c}_n\bar{x}_n \Rightarrow s_n) \\ & & \mid & \mathsf{C}[x\backslash s] \mid t[x\backslash \mathsf{C}] \end{array}$$

Definition 2.6. The **theory of sharing** λ_{sh} consists of the reduction rules over Λ_{sh} given below, closed by full contexts. We write $\to_{\mathsf{sh}}$ for the reduction relation.

$$\begin{array}{rcll} (\lambda x.t)\mathsf{L}\ s & \mapsto_{\mathtt{dB}} & t[x\backslash s]\mathsf{L} & \\ \mathsf{C}[\![x]\!][x\backslash v\mathsf{L}] & \mapsto_{\mathtt{lsv}} & \mathsf{C}[v][x\backslash v]\mathsf{L} & \\ t[x\backslash s] & \mapsto_{\mathtt{gc}} & t, & \text{if } x \notin \mathsf{fv}(t) \\ \mathtt{fix}(x.t) & \mapsto_{\mathtt{fix}} & t[x\backslash \mathtt{fix}(x.t)] & \\ \mathtt{case}\ \mathsf{A}[\mathbf{c}_j]\mathsf{L}\ \mathtt{of}\ (\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I} & \mapsto_{\mathtt{case}} & s_j[\bar{x}_j\backslash \mathsf{A}]\mathsf{L} & \\ & & & \text{if } |\mathsf{A}[\Box]| = |\bar{x}_j| \text{ and } j \in I \end{array}$$

The dB 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 $[x\backslash s]$. The lsv rule substitutes a free occurrence of x with the value v. Since variables in v 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

Figure 1 The set of $\to_{\mathsf{sh}}$-normal forms ($\mathbb{X} \in \{\mathcal{S}, \mathcal{L}, \mathcal{E}, \mathcal{K}\}$)

$$\frac{}{\mathbf{c} \in \mathcal{K}}\ \textsc{cNfCons} \qquad \frac{t \in \mathcal{K} \quad s \in \mathcal{N}}{t\,s \in \mathcal{K}}\ \textsc{cNfApp}$$

$$\frac{}{x \in \mathcal{S}}\ \textsc{sNfVar} \qquad \frac{t \in \mathcal{S} \quad s \in \mathcal{N}}{t\,s \in \mathcal{S}}\ \textsc{sNfApp}$$

$$\frac{t \in \mathcal{K} \cup \mathcal{L} \cup \mathcal{S} \quad t \not\succ (\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I} \quad (s_i \in \mathcal{N})_{i\in I}}{\mathtt{case}\ t\ \mathtt{of}\ (\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I} \in \mathcal{E}}\ \textsc{eNfStrt}$$

$$\frac{t \in \mathcal{E} \quad s \in \mathcal{N}}{t\,s \in \mathcal{E}}\ \textsc{eNfApp} \qquad \frac{t \in \mathcal{E} \quad (s_i \in \mathcal{N})_{i\in I}}{\mathtt{case}\ t\ \mathtt{of}\ (\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I} \in \mathcal{E}}\ \textsc{eNfCase}$$

$$\frac{t \in \mathcal{N}}{\lambda x.t \in \mathcal{L}}\ \textsc{lNfLam} \qquad \frac{t \in \mathbb{X} \quad s \in \mathcal{S} \cup \mathcal{E} \quad x \in \mathsf{fv}(t)}{t[x\backslash s] \in \mathbb{X}}\ \textsc{nfSub}$$

$$\frac{t \in \mathcal{K}}{t \in \mathcal{N}}\ \textsc{nfCons} \quad \frac{t \in \mathcal{S}}{t \in \mathcal{N}}\ \textsc{nfStruct} \quad \frac{t \in \mathcal{E}}{t \in \mathcal{N}}\ \textsc{nfError} \quad \frac{t \in \mathcal{L}}{t \in \mathcal{N}}\ \textsc{nfLam}$$

substitutions. Rule fix is standard. Rule case tests whether the condition of the case "matches" one of its branches. Note that the condition $\mathsf{A}[\mathbf{c}_j]\mathsf{L}$ may have explicit substitutions interspersed. The **length** of a constant context is defined as follows: $|\Box| := 0$ and $|\mathsf{L}[\mathsf{A}]\,t| := 1 + |\mathsf{A}|$. Given a list of variables $\bar{x}$ and a constant context A s.t. their lengths coincide, we define the substitution context $[\bar{x}\backslash\mathsf{A}]$ as follows: $[\epsilon\backslash\Box] := \Box$ and $[\bar{x}, y\backslash\mathsf{L}[\mathsf{A}]\,t] := [\bar{x}\backslash\mathsf{A}]\mathsf{L}[y\backslash t]$. The reduct of $\mapsto_{\mathtt{case}}$ uses this notion to build an appropriate list of explicit substitutions for each parameter of the branch.

Remark 4. *t definable in λ_{sh} iff $t \in \mathtt{WN}(\to_{\mathsf{sh}})$. This follows from confluence of $\to_{\mathsf{sh}}$.*

A characterization of the $\to_{\mathsf{sh}}$-normal forms is given in Fig. 1. The **normal form judgement** $t \in \mathcal{N}$ is defined simultaneously with four other judgements, namely **constant normal forms** $t \in \mathcal{K}$, **structure normal forms** $t \in \mathcal{S}$, **error normal forms** $t \in \mathcal{E}$, and **abstraction normal forms** $t \in \mathcal{L}$. We comment on some salient rules. First note that rule eNfStrt captures a blocked case where its condition is not a blocked case itself. If the condition of the case is $t \in \mathcal{L} \cup \mathcal{S}$, then we know that it cannot possibly match any branch. If $t \in \mathcal{K}$, we must make sure of this, as explained next. We say a term t **enables** a branch in a list of branches $(\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I}$, written $t \succ (\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I}$, if $t = \mathsf{A}[\mathbf{c}_j]\mathsf{L}$, for some A, L, and $j \in I$ and $|\mathsf{A}| = |\bar{x}_j|$. This is the natural extension of the notion of t matching a branch in λ_{e} but where t may be "polluted" with explicit substitutions. Note that if $t \not\succ (\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I}$, then either, 1) $t \neq \mathsf{A}[\mathbf{c}]\mathsf{L}$ for any $\mathsf{A}, \mathbf{c}, \mathsf{L}$; or 2) $t = \mathsf{A}[\mathbf{c}]\mathsf{L}$ with $\mathbf{c} \notin \{\mathbf{c}_i\}_{i\in I}$; or 3) $t = \mathsf{A}[\mathbf{c}]\mathsf{L}$ and $\mathbf{c} = \mathbf{c}_j$ for some $j \in I$ but $|\mathsf{A}| \neq |\bar{x}_j|$. Rule nfSub is actually a rule scheme in which $\mathbb{X}$ can be any of $\mathcal{S}$, $\mathcal{L}$, $\mathcal{E}$, or $\mathcal{K}$. Condition $x \in \mathsf{fv}(t)$ is required since we would otherwise have a gc-redex. Condition $s \in \mathcal{S} \cup \mathcal{E}$ is required too since we would otherwise have a lsv redex.

Lemma 2.7. *$t \in \mathcal{N}$ iff $t \in \mathsf{NF}(\to_{\mathsf{sh}})$.*

We conclude with a simple result that relates reduction in $\to_{\mathsf{sh}}$ with that in $\to_{\mathsf{e}}$.

Lemma 2.8. *Let $t, s \in \Lambda_{\mathsf{sh}}$.*

(1) *If $t \to_{\mathsf{sh}} s$, then $t^\diamond \twoheadrightarrow_{\mathsf{e}} s^\diamond$.*
(2) *$t \in \mathsf{NF}(\to_{\mathsf{sh}})$ implies $t^\diamond \in \mathsf{NF}(\to_{\mathsf{e}})$.*

3 INTERSECTION TYPES FOR THE THEORY OF SHARING

This section introduces $\mathcal{T}$, a non-idempotent intersection type system for λ_{sh}. We assume $\alpha, \beta, \gamma, \ldots$ to range over a set of **type variables**. The set of **types** is ranged over by $\tau, \sigma, \rho, \ldots$, and **finite multisets of types** are ranged over by $\mathcal{M}, \mathcal{N}, \mathcal{P}, \ldots$. The empty multiset is written $[]$, and $[\tau_1, \ldots, \tau_n]$ stands for the multiset containing each of the types τ_i with their corresponding multiplicities. Moreover, $\mathcal{M} + \mathcal{N}$ stands for the (additive) union of multisets. For instance $[\mathbf{a}, \mathbf{b}] + [\mathbf{b}, \mathbf{c}] = [\mathbf{a}, \mathbf{b}, \mathbf{b}, \mathbf{c}]$.

Definition 3.1. The types of $\mathcal{T}$ are defined as follows:

Types	τ	::=	$\alpha \mid \mathcal{M} \to \tau \mid D \mid E$
Datatypes	D	::=	$\mathbf{c} \mid D\,\mathcal{M}$
PreError type	G	::=	$\Bbbk\,\tau\,\bar{B} \mid G\,\tau$
Error types	E	::=	$\langle G \rangle \mid E\,\tau$
Branch type	B	::=	$\bar{\mathcal{M}} \Rightarrow \tau$

The type α is a type variable, $\mathcal{M} \to \tau$ is a **function type**, D is a *datatype* and E is an *error type*. A **datatype** is either a constant type $\mathbf{c}$ or an applied datatype $D\,\mathcal{M}$. Informally, $\mathbf{c}\mathcal{M}_1 \ldots \mathcal{M}_n$ 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** $\langle \Bbbk\,\tau\,(\bar{\mathcal{M}}_i \Rightarrow \sigma_i)_{i\in I} \rho_1 \ldots \rho_j \rangle\, \rho_{j+1} \ldots \rho_k$ is the type of a case expression:

(1) whose condition has type τ and branches type $\bar{\mathcal{M}}_i \Rightarrow \sigma_i$;
(2) which is stuck;
(3) which has been applied to arguments of type $\rho_1 \ldots \rho_j$; and
(4) which is expecting arguments of type $\rho_{j+1} \ldots \rho_k$.

We call $\Bbbk$ 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. $\mathrm{dom}\,\Gamma$ is the domain of Γ, namely the set of x s.t. $\Gamma(x) \neq []$.

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

Definition 3.2. The typing system $\mathcal{T}$ 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 τ under context Γ and error log Σ.
(2) **Multi-typing** $(\Gamma; \Sigma \vdash t : \mathcal{M})$
Term t has the types in $\mathcal{M}$ under context Γ and error log Σ.
(3) **Application** $(\tau \mathbin{@} \mathcal{M} \Rightarrow \sigma)$
A term of type τ may be applied to an argument that has all the types in $\mathcal{M}$, resulting in a term of type σ.
(4) **Matching** $(\tau\ \langle \bar{b} \rangle\ \Gamma; \Sigma, \sigma)$
Type τ might be the condition of a case with branches $\bar{b}$, which will result in a term of type σ, provided certain hypotheses Γ and error logs Σ, or else fail.

We write $\pi, \xi, \ldots$ for typing derivations and $\pi(\Gamma; \Sigma \vdash t : \tau)$ if π 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 tVar and the typing context to be empty in tCons. The error context, however, is said to be *intuitionistic* in that it may hold any number of error types. Rule tApp caters for typing applications of terms of functional type, data structures and error terms, to arguments by means of the *application judgement* $\tau \mathbin{@} \mathcal{M} \Rightarrow \sigma$. Indeed, τ may be of the form $\mathcal{M} \to \sigma$ (*cf.* tAppFun), or a datatype D in which case σ is $D\mathcal{M}$ (*cf.* tAppData), or an error type $\langle G \rangle\, \tau_1 \ldots \tau_n$ in which case $\mathcal{M}$ must be a singleton $[\tau_1]$ and σ of the $\langle G\,\tau_1 \rangle\, \tau_2 \ldots \tau_n$ (*cf.* 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 tFix splits its resources so that they are dealt out to be used for one unfolding (Γ) and the rest of the unfoldings (Δ). The tCase rule relies on the *matching judgement* $\tau\ \langle \bar{b} \rangle\ \Delta; \Sigma, \sigma$. The latter checks whether the type of the condition τ *matches* the list of branches. A **type** τ **matches** with a branch $\mathbf{c}\bar{x} \Rightarrow s$ if $\tau = \mathbf{c}\bar{\mathcal{M}}$ with $|\bar{\mathcal{M}}| = |\bar{x}|$. A type matches with a list of branches if it matches with at least one branch. Returning to our description of tCase, if τ matches with a branch, then that branch is typed (*cf.* tCMatch). However, if τ does not match any branch (*cf.* 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 $\langle \Bbbk\,\tau\,(\bar{\mathcal{M}}_i \Rightarrow \sigma_i)_{i\in I} \rangle\, \bar{\rho}$, 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 to. Finally, tMulti allows a term to be typed with a multiset type. In this rule, if $n = 0$, then $\sum_{i=1}^{n}[\tau_i]$ denotes the empty multiset $[]$.

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

3.1 An Example

Let t be the term $\mathtt{fix}(f.\lambda n.\mathtt{case}\ n\ \mathtt{of}\ \mathbf{z} \Rightarrow \mathbf{s}\,\mathbf{z}; \mathbf{s}\,n \Rightarrow \mathbf{s}\,n * f\,n)$ representing the factorial function. We exhibit type derivations for the judgements:

(1) $\emptyset; \emptyset \vdash t : [\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}]$; and
(2) $\emptyset; \emptyset \vdash t : [\mathbf{s}\,[\mathbf{z}, \mathbf{z}]] \to \mathbf{s}\,[\mathbf{z}]$.

We use $\bar{b}$ to denote the branches $\mathbf{z} \Rightarrow \mathbf{s}\,\mathbf{z}; \mathbf{s}\,n \Rightarrow \mathbf{s}\,n * f\,n$. The derivation π for the first judgement is in Fig. 3(a). Note the absence of f in the typing context of the judgement

$$\emptyset; \emptyset \vdash \lambda n.\mathtt{case}\ n\ \mathtt{of}\ \mathbf{z} \Rightarrow \mathbf{s}\,\mathbf{z}; \mathbf{s}\,n \Rightarrow \mathbf{s}\,n * f\,n : [\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}]$$

Since the type of n is $[\mathbf{z}]$ the branch with the recursive call will not be used and hence is not typed. The missing subderivation of π

Figure 2 Typing rules for $\mathcal{T}$

$$\frac{}{x : [\tau]; \Sigma \vdash x : \tau}\ \textsc{tVar} \qquad \frac{}{\varnothing; \Sigma \vdash \mathbf{c} : \mathbf{c}}\ \textsc{tCons}$$

$$\frac{\Gamma \oplus x :: \mathcal{M}; \Sigma \vdash t : \tau}{\Gamma; \Sigma \vdash \lambda x.t : \mathcal{M} \to \tau}\ \textsc{tAbs} \qquad \frac{\Gamma; \Sigma \vdash t : \tau \quad \tau \mathbin{@} \mathcal{M} \Rightarrow \sigma \quad \Delta; \Sigma \vdash s : \mathcal{M}}{\Gamma + \Delta; \Sigma \vdash ts : \sigma}\ \textsc{tApp}$$

$$\frac{\Gamma \oplus x :: \mathcal{M}; \Sigma \vdash t : \tau \quad \Delta; \Sigma \vdash \mathtt{fix}(x.t) : \mathcal{M}}{\Gamma + \Delta; \Sigma \vdash \mathtt{fix}(x.t) : \tau}\ \textsc{tFix} \qquad \frac{\Gamma; \Sigma \vdash t : \tau \quad \tau\ \langle \bar{b} \rangle\ \Delta; \Sigma, \sigma}{\Gamma + \Delta; \Sigma \vdash \mathsf{case}\ t\ \mathsf{of}\ \bar{b} : \sigma}\ \textsc{tCase}$$

$$\frac{\Gamma \oplus x :: \mathcal{M}; \Sigma \vdash t : \tau \qquad \Delta; \Sigma \vdash s : \mathcal{M}}{\Gamma + \Delta; \Sigma \vdash t[x \backslash s] : \tau}\ \textsc{tES} \qquad \frac{(\Gamma_i; \Sigma \vdash t : \tau_i)_{1 \le i \le n} \qquad (n \ge 0)}{\sum_{i=1}^{n} \Gamma_i; \Sigma \vdash t : \sum_{i=1}^{n} [\tau_i]}\ \textsc{tMulti}$$

$$\frac{}{\mathcal{M} \to \tau \mathbin{@} \mathcal{M} \Rightarrow \tau}\ \textsc{tAppFun} \qquad \frac{}{D \mathbin{@} \mathcal{M} \Rightarrow D\mathcal{M}}\ \textsc{tAppData} \qquad \frac{\bar{\tau} = \tau_1 \dots \tau_n}{\langle G \rangle\ \bar{\tau} \mathbin{@} [\tau_1] \Rightarrow \langle G\, \tau_1 \rangle\ \tau_2 \dots \tau_n}\ \textsc{tAppErr}$$

$$\frac{\mathbf{c}_j \bar{\mathcal{M}} \text{ matches } (\mathbf{c}_i \bar{x}_i \Rightarrow s_i)_{i \in I} \qquad \Gamma \oplus \bar{x}_j :: \bar{\mathcal{M}}; \Sigma \vdash s_j : \sigma_j}{\mathbf{c}_j \bar{\mathcal{M}}\ \langle (\mathbf{c}_i \bar{x}_i \Rightarrow s_i)_{i \in I} \rangle\ \Gamma; \Sigma, \sigma_j}\ \textsc{tCMatch}$$

$$\frac{\tau \text{ does not match } (\mathbf{c}_i \bar{x}_i \Rightarrow s_i)_{i \in I} \quad \left(\Gamma_i \oplus \bar{x}_i :: \bar{\mathcal{M}}_i; \Sigma \vdash s_i : \sigma_i\right)_{i \in I}}{\tau\ \langle (\mathbf{c}_i \bar{x}_i \Rightarrow s_i)_{i \in I} \rangle\ (\sum_{i \in I} \Gamma_i); \Sigma \cup \{\langle \mathbb{e}\, \tau\, (\bar{\mathcal{M}}_i \Rightarrow \sigma_i)_{i \in I} \rangle\ \bar{\rho}\}, \langle \mathbb{e}\, \tau\, (\bar{\mathcal{M}}_i \Rightarrow \sigma_i)_{i \in I} \rangle\ \bar{\rho}}\ \textsc{tCMismatch}$$

Figure 3 Example derivation

(a)
$$\dfrac{\dfrac{\dfrac{}{n : [\mathbf{z}]; \varnothing \vdash n : \mathbf{z}}\ \textsc{tVar} \qquad \dfrac{\pi_1}{\mathbf{z}\ \langle \bar{b} \rangle\ \varnothing; \varnothing, \mathbf{s}\,[\mathbf{z}]}}{n : [\mathbf{z}]; \varnothing \vdash \mathsf{case}\ n\ \mathsf{of}\ \bar{b} : \mathbf{s}\,[\mathbf{z}]}\ \textsc{tCase}}{\varnothing; \varnothing \vdash \lambda n.\mathsf{case}\ n\ \mathsf{of}\ \bar{b} : [\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}]}\ \textsc{tAbs} \qquad \dfrac{}{\varnothing; \varnothing \vdash t : []}\ \textsc{tMulti}$$
$$\varnothing; \varnothing \vdash t : [\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}]\ \textsc{tFix}$$

(b)
$$\dfrac{\dfrac{}{\varnothing; \varnothing \vdash \mathbf{s} : \mathbf{s}}\ \textsc{tCons} \qquad \dfrac{}{\mathbf{s} \mathbin{@} [\mathbf{z}] \Rightarrow \mathbf{s}\,[\mathbf{z}]}\ \textsc{tAppData} \qquad \dfrac{\dfrac{}{\varnothing; \varnothing \vdash \mathbf{z} : \mathbf{z}}\ \textsc{tCons}}{\varnothing; \varnothing \vdash \mathbf{z} : [\mathbf{z}]}\ \textsc{tMulti}}{\varnothing; \varnothing \vdash \mathbf{s}\,\mathbf{z} : \mathbf{s}\,[\mathbf{z}]}\ \textsc{tApp}$$
$$\mathbf{z}\ \langle (\mathbf{z} \Rightarrow \mathbf{s}\,\mathbf{z}; \mathbf{s}\,n \Rightarrow \mathbf{s}\,n * f\ n) \rangle\ \varnothing; \varnothing, \mathbf{s}\,[\mathbf{z}]\ \textsc{tCMatch}$$

(c)
$$\dfrac{\dfrac{\dfrac{}{n : [\mathbf{s}\,[\mathbf{z}, \mathbf{z}]]; \varnothing \vdash n : \mathbf{s}\,[\mathbf{z}, \mathbf{z}]}\ \textsc{tVar} \qquad \dfrac{\xi_1}{\mathbf{s}\,[\mathbf{z}, \mathbf{z}]\ \langle \bar{b} \rangle\ \{f : [[\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}]]\}; \varnothing, \mathbf{s}\,[\mathbf{z}]}}{f : [[\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}]], n : [\mathbf{s}\,[\mathbf{z}, \mathbf{z}]]; \varnothing \vdash \mathsf{case}\ n\ \mathsf{of}\ \bar{b} : \mathbf{s}\,[\mathbf{z}]}\ \textsc{tCase}}{f : [[\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}]]; \varnothing \vdash \lambda n.\mathsf{case}\ n\ \mathsf{of}\ \bar{b} : [\mathbf{s}\,[\mathbf{z}, \mathbf{z}]] \to \mathbf{s}\,[\mathbf{z}]}\ \textsc{tAbs} \qquad \dfrac{\pi}{\varnothing \vdash t : [[\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}]]}\ \textsc{tMulti}$$
$$\varnothing \vdash t : [\mathbf{s}\,[\mathbf{z}, \mathbf{z}]] \to \mathbf{s}\,[\mathbf{z}]\ \textsc{tFix}$$

(d)
$$\dfrac{\dfrac{\xi_2}{n : [\mathbf{z}]; \varnothing \vdash \mathbf{s}\,n : \mathbf{s}\,[\mathbf{z}]} \qquad \dfrac{f : [[\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}]]; \varnothing \vdash f : [\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}] \quad [\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}] \mathbin{@} [\mathbf{z}] \Rightarrow \mathbf{s}\,[\mathbf{z}] \quad n : [\mathbf{z}]; \varnothing \vdash n : [\mathbf{z}]}{n : [\mathbf{z}], f : [[\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}]]; \varnothing \vdash f\ n : \mathbf{s}\,[\mathbf{z}]}\ \textsc{tApp}}{f : [[\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}]], n : [\mathbf{z}, \mathbf{z}]; \varnothing \vdash \mathbf{s}\,n * f\ n : \mathbf{s}\,[\mathbf{z}]}\ \textsc{tProd}$$
$$\mathbf{s}\,[\mathbf{z}, \mathbf{z}]\ \langle \bar{b} \rangle\ \{f : [[\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}]]\}; \varnothing, \mathbf{s}\,[\mathbf{z}]\ \textsc{tCMatch}$$

called π_1, of the judgement $\mathbf{z}\ \langle \bar{b} \rangle\ \varnothing, \mathbf{s}\,[\mathbf{z}]$, is given in Fig 3(b). Since $\mathbf{z}$ matches with $(\mathbf{z} \Rightarrow \mathbf{s}\,\mathbf{z})$, we only type this branch.

A derivation ξ of $\varnothing \vdash t : [\mathbf{s}\,[\mathbf{z}, \mathbf{z}]] \to \mathbf{s}\,[\mathbf{z}]$ is in Fig. 3(c). We use the following typing rule for the product:

$$\frac{\Gamma;\Sigma \vdash t : \mathbf{s}^n\,\mathbf{z} \qquad \Delta;\Sigma \vdash s : \mathbf{s}^m\,\mathbf{z}}{\Gamma+\Delta;\Sigma \vdash t * s : \mathbf{s}^{n*m}\,\mathbf{z}}\ \textsc{tProd}$$

The derivation ξ_1 of $\mathbf{s}\,[\mathbf{z},\mathbf{z}]\ \langle \bar{b}\rangle\ \{f : [[\mathbf{z}] \to \mathbf{s}\,[\mathbf{z}]]\};\varnothing,\mathbf{s}\,[\mathbf{z}]$ is given in Fig. 3(d). The missing subderivation ξ_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 λ_{e}.

- **Step 1 (Sec. 4.1)**. Weakly normalising terms of λ_{e} are typable in the non-idempotent intersection type system $\mathcal{T}$.
- **Step 2 (Sec. 4.2).** Typable terms in $\mathcal{T}$ are weakly normalising in the theory of sharing λ_{sh}.
- **Step 3 (Sec. 6).** Factorization of the reduction sequence in λ_{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 λ_{e}, then the strategy computes a normal form u such that s is the unsharing of u.

4.1 Definable Terms in λ_{e} are Typable (Step 1)

This section addresses Step 1 of the diagram below, where $t \in \lambda_{\mathsf{e}}$:

$$t \in \mathsf{WN}(\lambda_{\mathsf{e}}) \overbrace{\Rightarrow}^{\textit{Step 1}} t \in \mathsf{Typable}(\mathcal{T}) \overbrace{\Rightarrow}^{\textit{Step 2}} t \in \mathsf{WN}(\lambda_{\mathsf{sh}})$$

As discussed in the Introduction, we don't want t to *just* be typable in $\mathcal{T}$ but to be typable with some additional *constraints* so that Step 2 holds too. For example, we mentioned that $x\,\Omega$ is typable by setting x to have type $[] \to \alpha$, for α a type variable; however, the term is not normalising in λ_{sh}. We must require that the typing judgement $\Gamma;\Sigma \vdash t : \tau$ be such that $[] \notin \mathcal{N}(\Gamma)$ and $[] \notin \mathcal{P}(\tau)$ [12]. Here $\mathcal{N}(\Gamma)$ and $\mathcal{P}(\tau)$ refer to the usual notions of negative and positive occurrences of types in τ (*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 $\mathsf{case}\ \mathbf{c}\ \mathsf{of}\ (\mathbf{d} \Rightarrow \mathbf{d})\,\Omega$. It is typable with, for example, type $\langle \mathbb{e}\,\mathbf{c}\,([\bar{\mathbf{d}}] \Rightarrow \mathbf{d})_{i\in I}[]\rangle$. 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 constraint to blocked case expressions.

Consider now the term $\mathsf{case}\ x\ \mathsf{of}\ (\mathbf{c} \Rightarrow \mathbf{d}) \cdot (\mathbf{e} \Rightarrow \Omega)$. This term is typable with type $\mathbf{d}$, if $x : [\mathbf{c}]$, however, it is not weakly normalizing in λ_{sh}. This motivates the new constraints $\mathbf{c} \notin \mathcal{P}(\Gamma)$, $\mathbf{c} \notin \mathcal{N}(\Sigma)$, and $\mathbf{c} \notin \mathcal{N}(\tau)$ in Def. 4.1. In particular, in a term such as $\mathsf{case}\ x\ \mathsf{of}\ (\mathbf{c} \Rightarrow \mathbf{d}) \cdot (\mathbf{e} \Rightarrow \mathbf{d})$ 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 Γ, we can place restrictions on their type through Γ. For example, in the above mentioned term $x\,\Omega$ one may require that $[] \notin \Gamma(x)$ to force the type system to account for Ω. However, now we may have term 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 τ, denoted $\mathcal{P}(\tau)$ (resp. $\mathcal{N}(\tau)$), is defined in Fig 4. A **type τ is good** if $\mathbf{c} \notin \mathcal{P}(\tau)$ and $[] \notin \mathcal{N}(\tau)$. We say $\mathcal{M}$ is good if each $\tau \in \mathcal{M}$ is good. A **typing context Γ is good** if $\Gamma = \Gamma_g\Gamma_e$ and $\forall x \in \mathrm{dom}\,\Gamma_g$, $\Gamma_g(x)$ is good and $\forall x \in \mathrm{dom}\,\Gamma_e$, $\Gamma_e(x)$ is an error type. A **typing judgement $\Gamma;\Sigma \vdash t : \tau$ is good** if

- Γ is good;
- $[] \notin \mathcal{P}(\Sigma)$ and $[] \notin \mathcal{P}(\tau)$;
- $\mathbf{c} \notin \mathcal{N}(\Sigma)$ and $\mathbf{c} \notin \mathcal{N}(\tau)$; and
- $\mathrm{covered}_{\Sigma}(\Gamma)$ and $\mathrm{covered}_{\Sigma}(\tau)$.

The proof of Step 1, namely that terms definable in λ_{e} terms are typable (Thm. 4.4), consists of two steps. First we show that $\to_{\mathsf{e}}$-normal forms are typable with good typing judgements. These typing judgements $\Gamma;\Sigma \vdash t : \tau$, for $t \in \mathsf{NF}(\to_{\mathsf{e}})$, are such that constants do not occur negatively in τ nor in Σ nor positively in Γ. However, constants may occur positively in τ such as when typing the normal form $\mathbf{c}$ and also negatively in Γ such as when typing the normal form $x\,\mathbf{c}$ where $x : [[\mathbf{c}] \to \alpha]$.

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

- *$x\,\bar{N}$, then τ is a type variable; and*
- *$(\mathsf{case}\ N_0\ \mathsf{of}\ (\mathbf{c}_i\bar{x}_i \Rightarrow N_i)_{i\in I})\bar{N}$, where N_0 does not match with $(\mathbf{c}_i\bar{x}_i \Rightarrow N_i)_{i\in I}$, then $\tau = \langle \mathbb{e}\,\tau\,(\bar{\mathcal{M}}_i \Rightarrow \sigma_i)_{i\in I}\,\rho_1 \dots\,\rho_k\rangle$ with $k = |\bar{N}|$.*

The second step consists of showing subject expansion for $\to_{\mathsf{e}}$ (*i.e.* $t \to_{\mathsf{e}} s$ and $\Gamma;\Sigma \vdash s : \tau$ imply $\Gamma;\Sigma \vdash t : \tau$).

Lemma 4.3 ($\to_{\mathsf{e}}$-expansion). *Let $t \to_{\mathsf{e}} s$. If $\Gamma;\Sigma \vdash s : \tau$ then $\Gamma;\Sigma \vdash t : \tau$.*

Theorem 4.4 (Step 1). *Suppose t is definable in λ_{e}. Then there exists a context Γ, an error context Σ, a type τ and a derivation π s.t. $\pi(\Gamma;\Sigma \vdash t : \tau)$ with $\Gamma;\Sigma \vdash t : \tau$ good.*

4.2 Typable Terms are Definable in λ_{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, lets 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 λ_{sh} (*i.e.* are in $\mathsf{NF}(\to_{\mathsf{sh}})$). That a redex is accounted for in a typing derivation π is expressed as the redex occurring at a *typed occurrence* in π. For example, in a term such as

Figure 4 Positive and negative types

$$
\begin{array}{rcl}
\mathcal{P}(\alpha) & := & \{\alpha\} \\
\mathcal{P}(\mathcal{M} \to \tau) & := & \mathcal{N}(\mathcal{M}) \cup \mathcal{P}(\tau) \cup \{\mathcal{M} \to \tau\} \\
\mathcal{P}(\mathbf{c}) & := & \{\mathbf{c}\} \\
\mathcal{P}(D\,\mathcal{M}) & := & \mathcal{P}(D) \cup \mathcal{P}(\mathcal{M}) \cup \{D\,\mathcal{M}\} \\
\mathcal{P}(E\,\tau) & := & \mathcal{P}(E) \cup \mathcal{P}(\tau) \cup \{E\,\tau\} \\
\mathcal{P}(\langle G \rangle) & := & \mathcal{P}(G) \cup \{G\} \\
\mathcal{P}(G\,\tau) & := & \mathcal{P}(G) \cup \mathcal{P}(\tau) \cup \{G\,\tau\} \\
\mathcal{P}(\mathbb{e}\,\tau\,\bar{B}) & := & \mathcal{P}(\tau) \cup \mathcal{P}(\bar{B}) \cup \{\mathbb{e}\,\tau\,\bar{B}\} \\
\mathcal{P}(\mathcal{M}_1, \ldots, \mathcal{M}_n \Rightarrow \tau) & := & \bigcup_{i \in 1..n} \mathcal{N}(\mathcal{M}_i) \cup \mathcal{P}(\tau) \cup \{\bar{\mathcal{M}} \Rightarrow \tau\} \\
\mathcal{P}(\mathcal{M}) & := & \bigcup_{\tau \in \mathcal{M}} \mathcal{P}(\tau) \cup \{\mathcal{M}\} \\
\mathcal{P}(\Gamma; \Sigma \vdash \tau) & := & \mathcal{N}(\Gamma) \cup \mathcal{P}(\Sigma) \cup \mathcal{P}(\tau) \\
\mathcal{P}(\Gamma) & := & \bigcup \mathcal{P}(\Gamma(x)), \text{ for all } x \in \operatorname{dom}\Gamma
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{N}(\alpha) & := & \varnothing \\
\mathcal{N}(\mathcal{M} \to \tau) & := & \mathcal{P}(\mathcal{M}) \cup \mathcal{N}(\tau) \\
\mathcal{N}(\mathbf{c}) & := & \varnothing \\
\mathcal{N}(D\,\mathcal{M}) & := & \mathcal{N}(D) \cup \mathcal{N}(\mathcal{M}) \\
\mathcal{N}(E\,\tau) & := & \mathcal{N}(E) \cup \mathcal{N}(\tau) \\
\mathcal{N}(\langle G \rangle) & := & \mathcal{N}(G) \\
\mathcal{N}(G\,\tau) & := & \mathcal{N}(G) \cup \mathcal{N}(\tau) \\
\mathcal{N}(\mathbb{e}\,\tau\,\bar{B}) & := & \mathcal{N}(\tau) \cup \mathcal{N}(\bar{B}) \\
\mathcal{N}(\mathcal{M}_1, \ldots, \mathcal{M}_n \Rightarrow \tau) & := & \bigcup_{i \in 1..n} \mathcal{P}(\mathcal{M}_i) \cup \mathcal{N}(\tau) \\
\mathcal{N}(\mathcal{M}) & := & \bigcup_{\tau \in \mathcal{M}} \mathcal{N}(\tau) \\
\mathcal{N}(\Gamma; \Sigma \vdash \tau) & := & \mathcal{P}(\Gamma) \cup \mathcal{N}(\Sigma) \cup \mathcal{N}(\tau) \\
\mathcal{N}(\Gamma) & := & \bigcup \mathcal{N}(\Gamma(x)), \text{ for all } x \in \operatorname{dom}\Gamma
\end{array}
$$

$$
\begin{array}{rcl}
\operatorname{covered}_\Sigma(\alpha)_T & := & \mathit{true} \\
\operatorname{covered}_\Sigma(\mathcal{M} \to \tau)_T & := & \operatorname{covered}_\Sigma(\mathcal{M})_T \wedge \operatorname{covered}_\Sigma(\tau)_T \\
\operatorname{covered}_\Sigma(D)_T & := & \operatorname{covered}_\Sigma(D)_D \\
\operatorname{covered}_\Sigma(E)_T & := & \operatorname{covered}_\Sigma(E)_E \wedge E \in \Sigma \\
\\
\operatorname{covered}_\Sigma(\mathbf{c})_D & := & \mathit{true} \\
\operatorname{covered}_\Sigma(D\,\mathcal{M})_D & := & \operatorname{covered}_\Sigma(D)_D \wedge \operatorname{covered}_\Sigma(\mathcal{M})_T
\end{array}
\qquad
\begin{array}{rcl}
\operatorname{covered}_\Sigma(\mathbb{e}\,\tau\,\bar{B})_G & := & \operatorname{covered}_\Sigma(\tau)_T \wedge \operatorname{covered}_\Sigma(\bar{B})_B \\
\operatorname{covered}_\Sigma(G\,\tau)_G & := & \operatorname{covered}_\Sigma(G)_G \wedge \operatorname{covered}_\Sigma(\tau)_T \\
\\
\operatorname{covered}_\Sigma(\langle G \rangle)_E & := & \operatorname{covered}_\Sigma(G)_G \\
\operatorname{covered}_\Sigma(E\,\tau)_E & := & \operatorname{covered}_\Sigma(E)_E \wedge \operatorname{covered}_\Sigma(\tau)_T \\
\\
\operatorname{covered}_\Sigma(\bar{\mathcal{M}} \Rightarrow \tau)_B & := & \operatorname{covered}_\Sigma(\bar{\mathcal{M}})_T \wedge \operatorname{covered}_\Sigma(\tau)_T \\
\\
\operatorname{covered}_\Sigma(\Gamma) & := & \bigwedge_{x \in \operatorname{dom}\Gamma} \operatorname{covered}_\Sigma(\Gamma(x))
\end{array}
$$

$x\,(\mathsf{id}\ \mathsf{id})$ with $x : [[] \to \alpha]$, the redex $\mathsf{id}\ \mathsf{id}$ is not typed-redex since there is no subderivation of π that types/accounts for it.

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 λ_{sh}.*

Assembling Step 1 and Step 2 we obtain:

Theorem 4.6 (Soundness of λ_{sh} w.r.t. λ_{e}). *Let t be a term in Λ_{e}. If $t \in \mathsf{WN}(\to_{\mathsf{e}})$, then $t \in \mathsf{WN}(\to_{\mathsf{sh}})$. More precisely, if $t \twoheadrightarrow_{\mathsf{e}} \mathsf{n}_{\mathsf{e}}$, where $\mathsf{n}_{\mathsf{e}} \in \Lambda_{\mathsf{sh}}$ is a $\to_{\mathsf{e}}$-normal form, then $t \twoheadrightarrow_{\mathsf{sh}} \mathsf{n}_{\mathsf{sh}}$, where n_{sh} is a $\to_{\mathsf{sh}}$-normal form. Moreover, $\mathsf{n}^{\diamond}_{\mathsf{sh}} = \mathsf{n}_{\mathsf{e}}$.*

Proof. Let $t \twoheadrightarrow_{\mathsf{e}} \mathsf{n}_{\mathsf{e}}$, where n_{e} is in $\to_{\mathsf{e}}$-nf. Then $\pi(\Gamma; \Sigma \vdash t : \tau)$ and $\Gamma; \Sigma \vdash \tau$ is good by Thm. 4.4. But then t is weakly $\to_{\mathsf{sh}}$-normalising by Thm. 4.5, so that $t \twoheadrightarrow_{\mathsf{sh}} \mathsf{n}_{\mathsf{sh}}$, where n_{sh} is in $\to_{\mathsf{sh}}$-nf. By Lem. 2.8(1) $t^{\diamond} \twoheadrightarrow_{\beta} \mathsf{n}^{\diamond}_{\mathsf{sh}}$ and by Lem. 2.8(2) $\mathsf{n}^{\diamond}_{\mathsf{sh}} \in \mathsf{NF}(\to_{\mathsf{e}})$. Since $t^{\diamond} = t \twoheadrightarrow_{\beta} \mathsf{n}_{\mathsf{e}}$ and $t^{\diamond} \twoheadrightarrow_{\mathsf{e}} \mathsf{n}^{\diamond}_{\mathsf{sh}}$, then we conclude $\mathsf{n}^{\diamond}_{\mathsf{sh}} = \mathsf{n}_{\mathsf{e}}$ because $\to_{\mathsf{e}}$ is Church-Rosser. □

5 THE STRONG CALL-BY-NEED STRATEGY

The strong call-by-need *strategy* $\rightarrowtail^{\vartheta}_{\mathsf{sh}}$ is a binary relation over terms in Λ_{sh} and is parameterized over a set ϑ 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 $\mathtt{C} \in C^{h}_{\vartheta}$ where $\mathtt{C}$ is a full context, ϑ is a set of variables and h is a symbol called *discriminator* of the context. This symbol may be one of '$\cdot$', 'λ' or any constant $\mathbf{c}, \mathbf{d}, \ldots$ 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 $\mathtt{C}$ such that the evaluation context judgement $\mathtt{C} \in C^{h}_{\vartheta}$ is derivable using the rules in Fig. 5.

eBox states that any redex at the root is needed (we may disregard ϑ and '$\cdot$' for now). Rule eApp-L allows reduction to take place to the left of an application. We must make sure that $\mathtt{C}$ is not an abstraction. This is achieved by requiring that $h \neq \lambda$ (*cf.* eLam and how all rules persist h). Rule eAppRStruct 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 $t\,\mathtt{C} \in C^{\cdot}_{\vartheta}$ reflects that t is not headed by a constant and that $t\,\mathtt{C}$ is not an abstraction. Rule eAppRCons is similar only that the discriminator is set to the head variable of t via $\mathsf{hc}(t)$ and will be consulted when deciding if reduction can take place in the condition of a case (*cf.* eCase1). This function is defined as: $\mathsf{hc}(\mathbf{c}) := \mathbf{c}$, $\mathsf{hc}(t\,s) := \mathsf{hc}(t)$ and $\mathsf{hc}(t[x\backslash s]) := \mathsf{hc}(t)$. Note that $\mathsf{hc}(\mathtt{A}[\mathbf{c}]\mathtt{L}) = \mathbf{c}$.

The role of frozen variables is best explained in the setting of eSubsLNonStruct and eSubsLStruct. In a term t such as $x[x\backslash y\,s]$, clearly $y\,s$ is not to be substituted for x since it is not an answer. Thus, computation has to proceed in s. However, if t is placed under an explicit substitution, then whether we should reduce s *depends* on its context. For example, we do want to reduce it in $x[x\backslash y\,s][y\backslash z]$ but not in $x[x\backslash y\,s][y\backslash \lambda z.\mathbf{c}]$ since $\lambda z.\mathbf{c}$ does not use s. These two examples motivate eSubsLStruct (z is a structure normal form) and eSubsLNonStruct ($\lambda z.\mathbf{c}$ 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 $y\,s$, we must know that y will never be substituted for, or else, that it is *frozen*. Rule eSubs-R allows computation to take place in the body of an explicit substitution.

There is no rule for $\mathtt{fix}(x.t)$ 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.* eCase1). This is achieved by requiring that the discriminator either is not a constant listed in the branches ($h \notin \{\mathbf{c}_i\}_{i \in I}$) or that, if it is, then the number of expected arguments by the branch are not met ($|\mathtt{C}[y]| \neq |\bar{x_j}|$). The notation $|\mathtt{C}[y]|$ counts the number of arguments in the spine of the term $\mathtt{C}[y]$. It is defined as follows:

Figure 5 Evaluation Contexts

$$\frac{}{\Box \in C^{\cdot}_{\vartheta}}\ \textsc{eBox} \qquad \frac{\mathtt{C} \in C^{h}_{\vartheta} \quad h \neq \lambda}{\mathtt{C}\ t \in C^{h}_{\vartheta}}\ \textsc{eAppL} \qquad \frac{t \in \mathcal{S}_{\vartheta} \cup \mathcal{E}_{\vartheta} \quad \mathtt{C} \in C^{h}_{\vartheta}}{t\ \mathtt{C} \in C^{\cdot}_{\vartheta}}\ \textsc{eAppRStruct} \qquad \frac{t \in \mathcal{K}_{\vartheta} \quad \mathtt{C} \in C^{h}_{\vartheta}}{t\ \mathtt{C} \in C^{\mathrm{hc}(t)}_{\vartheta}}\ \textsc{eAppRCons}$$

$$\frac{\mathtt{C} \in C^{h}_{\vartheta} \quad t \notin \mathcal{S}_{\vartheta} \cup \mathcal{E}_{\vartheta} \quad x \notin \vartheta}{\mathtt{C}[x\backslash t] \in C^{h}_{\vartheta}}\ \textsc{eSubsLNonStruct} \qquad \frac{\mathtt{C} \in C^{h}_{\vartheta \cup \{x\}} \quad t \in \mathcal{S}_{\vartheta} \cup \mathcal{E}_{\vartheta}}{\mathtt{C}[x\backslash t] \in C^{h}_{\vartheta}}\ \textsc{eSubsLStruct}$$

$$\frac{\mathtt{C}_1 \in C^{h}_{\vartheta} \quad \mathtt{C}_2 \in C^{\cdot}_{\vartheta}}{\mathtt{C}_1[\![x]\!][x\backslash \mathtt{C}_2] \in C^{h}_{\vartheta}}\ \textsc{eSubsR} \qquad \frac{\mathtt{C} \in C^{h}_{\vartheta \cup \{x\}}}{\lambda x.\mathtt{C} \in C^{\lambda}_{\vartheta}}\ \textsc{eLam}$$

$$\frac{\mathtt{C} \in C^{h}_{\vartheta} \quad h \notin \{\mathbf{c}_i\}_{i\in I} \text{ or } h = \mathbf{c}_j \in \{\mathbf{c}_i\}_{i\in I} \text{ and } |\mathtt{C}[y]| \neq |\bar{x_j}|}{\mathtt{case}\ \mathtt{C}\ \mathtt{of}\ (\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I} \in C^{\cdot}_{\vartheta}}\ \textsc{eCase1} \qquad \frac{t \in \mathcal{N}_{\vartheta} \quad t \not\succ (\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I} \quad t_k \in \mathcal{N}_{\vartheta \cup \bar{x_k}} \text{ for all } k < j \quad \mathtt{C} \in C^{h}_{\vartheta \cup \bar{x_i}}}{\mathtt{case}\ t\ \mathtt{of}\ \mathbf{c}_1\bar{x_1} \Rightarrow t_1, \ldots, \mathbf{c}_j\bar{x_j} \Rightarrow \mathtt{C}, \ldots, \mathbf{c}_n\bar{x_n} \Rightarrow t_n \in C^{\cdot}_{\vartheta}}\ \textsc{eCase2}$$

$$\begin{array}{rcl} |x| & := & 0 \\ |\mathbf{c}| & := & 0 \\ |\lambda x.t| & := & 0 \\ |t\ s| & := & 1 + |t| \end{array} \qquad \begin{array}{rcl} |\mathtt{fix}(x.t)| & := & 0 \\ |t[x\backslash s]| & := & |t| \\ |\mathtt{case}\ t\ \mathtt{of}\ \bar{b}| & := & 0 \end{array}$$

We know that in fact $\mathtt{C}[y]$ is a constant answer:

Lemma 5.2 (Answer contexts are answers). *Suppose* $\mathtt{C} \in C^{h}_{\vartheta}$.

- *If* $h = \mathbf{c}$*, then, for any term* t*, there exist* $\mathtt{A}$ *and* $\mathtt{L}$ *s.t.* $\mathtt{C}[t] = \mathtt{A}[\mathbf{c}]\mathtt{L}$.
- *If* $h = \lambda$*, then, for any term* t*, there exists a variable* x*, term* s *and substitution context* $\mathtt{L}$ *s.t.* $\mathtt{C}[t] = (\lambda x.s)\mathtt{L}$*. Moreover,* $\mathtt{C}$ *is either of the form* $(\lambda x.\mathtt{C}')\mathtt{L}$ *or* $(\lambda x.t)\mathtt{L}_1[y\backslash \mathtt{C}']\mathtt{L}_2$.

For reduction to proceed in a branch j (*cf.* eCase2), the condition must be in normal form, each branch i with $i \in 1..j$ must be in normal form and the condition must not enable any branch ($t \not\succ (\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I}$). 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 $\rightarrowtail^{\vartheta}_{\mathsf{sh}}$ strategy is defined by the following rules.

$$\begin{array}{rcll} \mathtt{C}[(\lambda x.t)\mathtt{L}\ s] & \rightarrowtail^{\vartheta}_{\mathsf{sh}} & \mathtt{C}[t[x\backslash s]\mathtt{L}] & (\mathsf{dB}) \\ & & \text{if } \mathtt{C} \in C^{h}_{\vartheta} & \\ \mathtt{C}_1[\mathtt{C}_2[\![x]\!][x\backslash v\mathtt{L}]] & \rightarrowtail^{\vartheta}_{\mathsf{sh}} & \mathtt{C}_1[\mathtt{C}_2[v][x\backslash v]\mathtt{L}] & (\mathtt{lsv}) \\ & & \text{if } \mathtt{C}_1[\mathtt{C}_2[\Box][x\backslash v\mathtt{L}]] \in C^{h}_{\vartheta} & \\ \mathtt{C}[\mathtt{fix}(x.t)] & \rightarrowtail^{\vartheta}_{\mathsf{sh}} & \mathtt{C}[t[x\backslash \mathtt{fix}(x.t)]] & (\mathtt{fix}) \\ \mathtt{C}[\mathtt{case}\ \mathtt{A}[\mathbf{c}_j]\mathtt{L}\ \mathtt{of}\ (\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I}] & \rightarrowtail^{\vartheta}_{\mathsf{sh}} & \mathtt{C}[s_j[\bar{x}_j\backslash \mathtt{A}]\mathtt{L}] & (\mathtt{case}) \\ & & \text{if } \mathtt{C} \in C^{h}_{\vartheta} \text{ and } j \in I \text{ and } |\mathtt{A}[\Box]| = |\bar{x}_j| & \end{array}$$

The discriminator h in the conditions of all rules is existentially quantified. The condition $\mathtt{C}_1[\mathtt{C}_2[\Box][x\backslash v\mathtt{L}]] \in C^{h}_{\vartheta}$ in the definition of the `lsv`-redex carries over from [9]. It avoids `lsv`-reducing $(\lambda x.y)[y\backslash \mathtt{id}]$ in $(\lambda x.y)[y\backslash \mathtt{id}]t$ so that the outermost `dB`-reduction step takes precedence instead. The condition also avoids to `lsv`-reduce $x[x\backslash(\lambda y.yz)[z\backslash \mathtt{id}]]$ on the variable z, so that the `lsv`-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* $\mathtt{C}[r]$*, if it is a* `dB`*-redex, a* `fix`*-redex, a* `case`*-redex or a variable bound to an answer. If* $\mathtt{C}_1[r_1] = \mathtt{C}_2[r_2]$*, where* $\mathtt{C}_1, \mathtt{C}_2 \in C^{h}_{\vartheta}$ *and* r_1, r_2 *are anchors, then* $\mathtt{C}_1 = \mathtt{C}_2$ *and* $r_1 = r_2$.

5.1 Normal Forms of the Strategy

We present an inductive characterization of the normal forms of $\rightarrowtail^{\vartheta}_{\mathsf{sh}}$. Since reduction in $\rightarrowtail^{\vartheta}_{\mathsf{sh}}$ is parameterized over a set of frozen variables ϑ, the normal forms too will be parameterized by this set. The set of **normal forms** over ϑ ($\mathcal{N}_{\vartheta}$) is comprised of the **constant normal forms** over ϑ ($\mathcal{K}_{\vartheta}$), the **structure normal forms** over ϑ ($\mathcal{S}_{\vartheta}$), the **error normal forms** over ϑ ($\mathcal{E}_{\vartheta}$) and the **lambda normal forms** over ϑ ($\mathcal{L}_{\vartheta}$). They are defined in Fig. 6 and are similar to the characterization of the $\rightarrow_{\mathsf{sh}}$-normal forms (Fig. 1) except that: 1) the set of frozen variables is tracked, 2) rule nfSub is refined into rules nfSubNG, and 3) a new rule nfSubG is added due to the absence of gc in $\rightarrowtail^{\vartheta}_{\mathsf{sh}}$. In rules nfSubNG and nfSubG, the symbol $\mathbb{X}$ represents either $\mathcal{S}_{\vartheta}$, $\mathcal{E}_{\vartheta}$, $\mathcal{L}_{\vartheta}$ or $\mathcal{K}_{\vartheta}$. Rule nfSubG helps capture terms such as $z[y\backslash x][x\backslash s]$. Note that $x \in \mathrm{fv}(z[y\backslash x])$ 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 $\mathrm{ngv}(t)$, are defined below[1], where $\bar{b}$ stands for $(\mathbf{c}_i\bar{x}_i \Rightarrow s_i)_{i\in I}$.

$$\begin{array}{rcl} \mathrm{ngv}(x) & := & \{x\} \\ \mathrm{ngv}(\lambda x.t) & := & \mathrm{ngv}(t) \setminus \{x\} \\ \mathrm{ngv}(ts) & := & \mathrm{ngv}(t) \cup \mathrm{ngv}(s) \\ \mathrm{ngv}(\mathtt{fix}(x.t)) & := & \mathrm{ngv}(t) \setminus \{x\} \\ \mathrm{ngv}(\mathbf{c}) & := & \varnothing \\ \mathrm{ngv}(\mathtt{case}\ t\ \mathtt{of}\ \bar{b}) & := & \mathrm{ngv}(t) \cup \bigcup_{i\in 1..n} \mathrm{ngv}(s_i) \setminus \bar{x}_i \\ \mathrm{ngv}(t[x\backslash s]) & := & (\mathrm{ngv}(t) \setminus \{x\}) \cup \begin{cases} \mathrm{ngv}(s) & \text{if } x \in \mathrm{ngv}(t) \\ \varnothing & \text{otherwise} \end{cases} \end{array}$$

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

Lemma 5.6. $\mathsf{NF}(\rightarrowtail^{\vartheta}_{\mathsf{sh}}) = \mathcal{N}_{\vartheta}$.

Figure 6 ϑ-normal forms of the strategy ($\mathbb{X} \in \{\mathcal{S}_\vartheta, \mathcal{L}_\vartheta, \mathcal{E}_\vartheta, \mathcal{K}_\vartheta\}$)

$$\frac{}{\mathbf{c} \in \mathcal{K}_\vartheta}\ \textsc{cNfCons} \qquad \frac{t \in \mathcal{K}_\vartheta \quad s \in \mathcal{N}_\vartheta}{t\,s \in \mathcal{K}_\vartheta}\ \textsc{cNfApp} \qquad \frac{x \in \vartheta}{x \in \mathcal{S}_\vartheta}\ \textsc{sNfVar} \qquad \frac{t \in \mathcal{S}_\vartheta \quad s \in \mathcal{N}_\vartheta}{t\,s \in \mathcal{S}_\vartheta}\ \textsc{sNfApp}$$

$$\frac{t \in \mathcal{K}_\vartheta \cup \mathcal{L}_\vartheta \cup \mathcal{S}_\vartheta \quad t \not\succ (\mathbf{c}_i \bar{x}_i \Rightarrow s_i)_{i\in I} \quad (s_i \in \mathcal{N}_{\vartheta \cup \bar{x}_i})_{i\in I}}{\mathsf{case}\ t\ \mathsf{of}\ (\mathbf{c}_i \bar{x}_i \Rightarrow s_i)_{i\in I} \in \mathcal{E}_\vartheta}\ \textsc{eNfStrt}$$

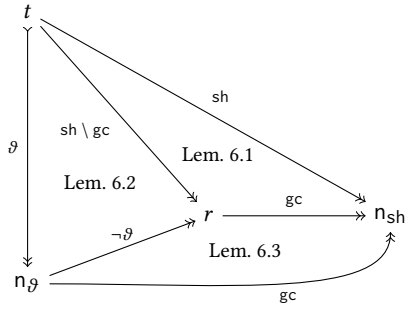
$$\frac{t \in \mathcal{E}_\vartheta \quad s \in \mathcal{N}_\vartheta}{t\,s \in \mathcal{E}_\vartheta}\ \textsc{eNfApp} \qquad \frac{t \in \mathcal{E}_\vartheta \quad (s_i \in \mathcal{N}_{\vartheta \cup \bar{x}_i})_{i\in I}}{\mathsf{case}\ t\ \mathsf{of}\ (\mathbf{c}_i \bar{x}_i \Rightarrow s_i)_{i\in I} \in \mathcal{E}_\vartheta}\ \textsc{eNfCase} \qquad \frac{t \in \mathcal{N}_{\vartheta \cup \{x\}}}{\lambda x.t \in \mathcal{L}_\vartheta}\ \textsc{lNfLam}$$

$$\frac{t \in \mathbb{X}_{\vartheta \cup \{x\}} \quad s \in \mathcal{S}_\vartheta \cup \mathcal{E}_\vartheta \quad x \in \mathsf{ngv}(t)}{t[x\backslash s] \in \mathbb{X}_\vartheta}\ \textsc{nfSubNG} \qquad \frac{t \in \mathbb{X}_\vartheta \quad x \notin \mathsf{ngv}(t)}{t[x\backslash s] \in \mathbb{X}_\vartheta}\ \textsc{nfSubG}$$

$$\frac{t \in \mathcal{K}_\vartheta}{t \in \mathcal{N}_\vartheta}\ \textsc{nfCons} \qquad \frac{t \in \mathcal{S}_\vartheta}{t \in \mathcal{N}_\vartheta}\ \textsc{nfStruct} \qquad \frac{t \in \mathcal{L}_\vartheta}{t \in \mathcal{N}_\vartheta}\ \textsc{nfLam} \qquad \frac{t \in \mathcal{E}_\vartheta}{t \in \mathcal{N}_\vartheta}\ \textsc{nfError}$$

6 A STANDARDIZATION THEOREM FOR THE THEORY OF SHARING

This section addresses a standardization theorem for λ_{sh}. Suppose t is definable in λ_{sh} as $\mathsf{n}_{\mathsf{sh}} \in \mathcal{N}$. Then there is a reduction sequence $t \twoheadrightarrow_{\mathsf{sh}} \mathsf{n}_{\mathsf{sh}}$ (*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 $t \twoheadrightarrow_{\mathsf{sh}} \mathsf{n}_{\mathsf{sh}}$ so that it can be factored into two parts $t \twoheadrightarrow^{\vartheta}_{\mathsf{sh}} u \twoheadrightarrow^{\neg\vartheta}_{\mathsf{sh}} \mathsf{n}_{\mathsf{sh}}$. The prefix $t \twoheadrightarrow^{\vartheta}_{\mathsf{sh}} u$ is reduction via the strong-call-by-need strategy; the double headed arrow indicates multiple steps of the strategy. The suffix $u \twoheadrightarrow^{\neg\vartheta}_{\mathsf{sh}} \mathsf{n}_{\mathsf{sh}}$ 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 $\rightarrow_{\mathsf{sh}}$-reduction step**, denoted $\rightarrow_{\mathsf{sh}\backslash\mathsf{gc}}$, is a $\rightarrow_{\mathsf{sh}}$-reduction step without using the $\mapsto_{\mathsf{gc}}$-rule.

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

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

Lemma 6.2 (Factorization of Strict Steps). *Let $\mathsf{fv}(t) \subseteq \vartheta$. If $t \twoheadrightarrow_{\mathsf{sh}\backslash\mathsf{gc}} u$, then there is a term s such that $t \twoheadrightarrow^{\vartheta}_{\mathsf{sh}} u \twoheadrightarrow^{\neg\vartheta}_{\mathsf{sh}} s$.*

Part III. As mentioned, if the $\rightarrow_{\mathsf{sh}}$ reduction sequence reaches a $\rightarrow_{\mathsf{sh}}$-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 ϑ, t be such that $\mathsf{fv}(t) \subseteq \vartheta$.*

1. *If $t \rightarrow_{\mathsf{gc}} \mathsf{n}_\vartheta$ with $\mathsf{n}_\vartheta \in \mathsf{NF}(\rightarrowtail^{\vartheta}_{\mathsf{sh}})$ then $t \in \mathsf{NF}(\rightarrowtail^{\vartheta}_{\mathsf{sh}})$.*
2. *If $t \rightarrow^{\neg\vartheta}_{\mathsf{sh}} \mathsf{n}_\vartheta$ with $\mathsf{n}_\vartheta \in \mathsf{NF}(\rightarrowtail^{\vartheta}_{\mathsf{sh}})$ then $t \in \mathsf{NF}(\rightarrowtail^{\vartheta}_{\mathsf{sh}})$ and there is u such that $t \twoheadrightarrow_{\mathsf{gc}} u$ and $\mathsf{n}_\vartheta \twoheadrightarrow_{\mathsf{gc}} u$.*

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

Theorem 6.4 (Standardization for $\rightarrow_{\mathsf{sh}}$). *Let ϑ, t be such that $\mathsf{fv}(t) \subseteq \vartheta$. If $t \twoheadrightarrow_{\mathsf{sh}} \mathsf{n}_{\mathsf{sh}}$, where $\mathsf{n}_{\mathsf{sh}} \in \mathsf{NF}(\rightarrow_{\mathsf{sh}})$, then there exists a term $\mathsf{n}_\vartheta \in \mathsf{NF}(\rightarrowtail^{\vartheta}_{\mathsf{sh}})$ such that $t \twoheadrightarrow^{\vartheta}_{\mathsf{sh}} \mathsf{n}_\vartheta$ and $\mathsf{n}_\vartheta \twoheadrightarrow_{\mathsf{gc}} \mathsf{n}_{\mathsf{sh}}$.*

Corollary 6.5 (Completeness of $\rightarrowtail^{\vartheta}_{\mathsf{sh}}$). *Let ϑ, t be such that $\mathsf{fv}(t) \subseteq \vartheta$. If $t \twoheadrightarrow_{\mathsf{e}} \mathsf{n}_{\mathsf{e}}$, where $\mathsf{n}_{\mathsf{e}} \in \mathsf{NF}(\rightarrow_{\mathsf{e}})$, then there exists a term $\mathsf{n}_\vartheta \in \mathsf{NF}(\rightarrowtail^{\vartheta}_{\mathsf{sh}})$ such that $t \twoheadrightarrow^{\vartheta}_{\mathsf{sh}} \mathsf{n}_\vartheta$ and $\mathsf{n}^{\diamond}_\vartheta = \mathsf{n}_{\mathsf{e}}$.*

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

[1] They may alternatively be characterized as $\mathsf{ngv}(t) = \mathsf{fv}(\downarrow_{\mathsf{gc}}(t))$, where $\downarrow_{\mathsf{gc}}(t)$ simply removes all garbage substitutions [9].

for call-by-need was studied in [24]. Completeness 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 combinator 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

[1] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2014. Distilling abstract machines. In *ICFP'14, Gothenburg, Sweden, September 1-3, 2014.* ACM, 363–376.

[2] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2015. A Strong Distillery. In *APLAS'15, Pohang, South Korea, November 30 - December 2, 2015. (LNCS)*, Xinyu Feng and Sungwoo Park (Eds.), Vol. 9458. Springer Verlag, 231–250.

[3] Beniamino Accattoli and Bruno Barras. 2017. The Negligible and Yet Subtle Cost of Pattern Matching. In *APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Bor-Yuh Evan Chang (Ed.), Vol. 10695. Springer, 426–447.

[4] Beniamino Accattoli and Claudio Sacerdoti Coen. 2017. On the value of variables. *Inf. Comput.* 255 (2017), 224–242. https://doi.org/10.1016/j.ic.2017.01.003

[5] Beniamino Accattoli and Giulio Guerrieri. 2016. Open Call-by-Value. In *APLAS'16, Hanoi, Vietnam, November 21-23, 2016. (LNCS)*, Atsushi Igarashi (Ed.), Vol. 10017. Springer, 206–226.

[6] Beniamino Accattoli and Delia Kesner. 2010. The Structural *lambda*-Calculus. In *CSL'10, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. (LNCS)*, Anuj Dawar and Helmut Veith (Eds.), Vol. 6247. Springer Verlag, 381–395.

[7] Zena M. Ariola and Matthias Felleisen. 1997. The Call-By-Need Lambda Calculus. *Journal of Functional Programming* 7, 3 (1997), 265–301.

[8] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. 1995. The Call-by-Need Lambda Calculus. In *POPL'95, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 233–246.

[9] Thibaut Balabonski, Pablo Barenbaum, Eduardo Bonelli, and Delia Kesner. 2017. Foundations of strong call by need. *PACMPL* 1, ICFP (2017), 20:1–20:29.

[10] Alexis Bernadet. 2014. *Types intersections non-idempotents pour rafiner la normalisation forte avec des informations quantitatives.* Ph.D. Dissertation. École Polytechnique.

[11] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. 2011. Full Reduction at Full Throttle. In *CPP'11, Kenting, Taiwan, December 7-9, 2011. (LNCS)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.), Vol. 7086. Springer, 362–377.

[12] Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. 2017. Non-idempotent intersection types for the Lambda-Calculus. *Logic Journal of the IGPL* 25, 4 (2017), 431–464.

[13] Stephen Chang and Matthias Felleisen. 2012. The Call-by-Need Lambda Calculus, Revisited. In *ESOP'12, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. (LNCS)*, Helmut Seidl (Ed.), Vol. 7211. Springer Verlag, 128–147.

[14] Thierry Coquand. 1996. An Algorithm for Type-Checking Dependent Types. *Sci. Comput. Program.* 26, 1-3 (1996), 167–177.

[15] Pierre Crégut. 1990. An Abstract Machine for Lambda-Terms Normalization. In *LISP and Functional Programming.* ACM, 333–340.

[16] Pierre Crégut. 2007. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation* 20, 3 (2007), 209–230.

[17] Daniel de Carvalho. 2009. Execution Time of lambda-Terms via Denotational Semantics and Intersection Types. *CoRR* abs/0905.4251 (2009), 1–36.

[18] Thomas Ehrhard and Laurent Regnier. 2006. Böhm Trees, Krivine's Machine and the Taylor Expansion of Lambda-Terms. In *CiE'06, Swansea, UK, June 30-July 5, 2006. (LNCS)*, Vol. 3988. Springer Verlag, 186–197.

[19] Álvaro García-Pérez, Pablo Nogueira, and Juan José Moreno-Navarro. 2013. Deriving the full-reducing Krivine machine from the small-step operational semantics of normal order. In *PPDP '13, Madrid, Spain, September 16-18, 2013.* ACM, 85–96.

[20] Philippa Gardner. 1994. Discovering Needed Reductions Using Type Theory. In *TACS'94 (LNCS)*, Masami Hagiya and John C. Mitchell (Eds.), Vol. 789. Springer Verlag, 555–574.

[21] Benjamin Grégoire and Xavier Leroy. 2002. A compiled implementation of strong reduction. In *ICFP '02, Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 235–246.

[22] Peter Henderson and James H. Morris. 1976. A Lazy Evaluator. In *POPL'76, Atlanta, Georgia, USA, January 1976*, Susan L. Graham, Robert M. Graham, Michael A. Harrison, William I. Grosky, and Jeffrey D. Ullman (Eds.). ACM Press, 95–103.

[23] Delia Kesner. 2016. Reasoning About Call-by-need by Means of Types. In *FOSSACS'16, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016. (LNCS)*, Bart Jacobs and Christof Löding (Eds.), Vol. 9634. Springer Verlag, 424–441.

[24] John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *POPL'93, Charleston, South Carolina, USA, January 1993.* ACM Press, 144–154.

[25] John Maraist, Martin Odersky, and Philip Wadler. 1998. The Call-by-Need Lambda Calculus. *Journal of Functional Programming* 8, 3 (1998), 275–317.

[26] Robin Milner. 2007. Local Bigraphs and Confluence: Two Conjectures: (Extended Abstract). *Electronic Notes in Theoretical Computer Science* 175, 3 (2007), 65–73.

[27] Christopher P. Wadsworth. 1971. *Semantics and Pragmatics of the Lambda Calculus.* Ph.D. Dissertation. Oxford University.