Useful Open Call-By-Need

Beniamino Accattoli, Maico Leberle

Useful Open Call-By-Need

Beniamino Accattoli
Inria & École Polytechnique, Palaiseau, France

Maico Leberle
Inria & École Polytechnique, Palaiseau, France

Abstract

This paper studies useful sharing, which is a sophisticated optimization for λ-calculi, in the context of call-by-need evaluation in presence of open terms. Useful sharing turns out to be harder in call-by-need than in call-by-name or call-by-value, because call-by-need evaluates inside environments, making it harder to specify when a substitution step is useful. We isolate the key involved concepts and prove the correctness and the completeness of useful sharing in this setting.

2012 ACM Subject Classification Theory of computation → Lambda calculus; Theory of computation → Operational semantics

Keywords and phrases lambda calculus, call-by-need, operational semantics, sharing, cost models

Digital Object Identifier 10.4230/LIPIcs.CSL.2022.4

Related Version There is an extended version with proofs.
Full Version: `https://arxiv.org/abs/2107.06591`

1 Introduction

Despite decades of research on how to best evaluate λ-terms, the topic is still actively studied and recent years have actually seen a surge in new results and sophisticated techniques. This paper is an attempt at harmonizing two of them, namely, *strong call-by-need* and *useful sharing*, under the influence of a third recently identified setting, *open call-by-value*. To describe our results, we have to first outline each of these approaches.

Call-by-Need. Call-by-need (shortened to CbNeed) is an evaluation scheme for the λ-calculus introduced in 1971 by Wadsworth [53] as an optimization of call-by-name (CbN), and nowadays lying at the core of the Haskell programming language. In the '90s, it was reformulated as operational semantics by Launchbury [45], Ariola and Felleisen [19], and Maraist et al. [48], and implemented by Sestoft [52] and further studied by Kutzner and Schmidt-Schauß [44]. Despite being decades old, CbNeed is still actively studied, perhaps more than ever before. The last decade indeed saw a number of studies by e.g. Ariola et al. [20], Chang and Felleisen [31], Danvy and Zerny [35], Downen et al. [36], Garcia et al. [37], Hackett and Hutton [39], Pédrot and Saurin [50], Mizuno and Sumii [49], Herbelin and Miquey [40], and Kesner et al. [42], plus those mentioned in the following paragraphs.

In the untyped, effect-free setting of the λ-calculus, CbNeed can be seen as borrowing the best aspects of call-by-value (CbV), of which it takes efficiency, and of CbN, of which it retains the better terminating behavior, as stressed in particular by Accattoli et al. [17]. In contrast to CbN and CbV, however, CbNeed cannot easily be managed at the small-step level of the usual operational semantics of the λ-calculus, based on β-reduction and meta-level substitution. Its fine dynamics, indeed, requires a decomposition of the substitution process acting on single variable occurrences at a time – what we refer to as *micro-step (operational) semantics* – and enriching λ-terms with some form of first-class *sharing*. While Wadsworth's original presentation is quite difficult to manage, along the years presentations of CbNeed have improved considerably ([45, 48, 19, 31]), up to obtaining neat definitions, as the one by

Accattoli et al. [6] (2014) in the *linear substitution calculus* (shortened to LSC), which led to elegant proofs of its correctness with respect to CbN, as done by Kesner [41] (2016), and of its relationship with neededness from a rewriting point of view, by Kesner et al. [43] (2018).

Strong Call-by-Need. Being motivated by functional languages, CbNeed is usually studied considering two restrictions with respect to the ordinary λ-calculus: 1) terms are closed, and 2) abstraction bodies are not evaluated. Let us call this setting *Closed CbNeed*. Extensions of CbNeed removing both these restrictions have been considered, obtaining what we shall refer to as *Strong CbNeed*. In his PhD thesis [27] (1999), Barras designs and implements an abstract machine for Strong CbNeed, which has then been used in the kernel of the Coq proof assistant to decide the convertibility of terms. Balabonski et al. [23] (2017) give instead the first formal operational semantics of Strong CbNeed, proving it correct with respect to Strong CbN– see also Barenbaum et al. [25], where the semantics of [23] is extended towards Barras's work; Biernacka and Charatonik [28], where it is studied via an abstract machine; Balabonski et al. [24] where it has recently been revisited and partially formalized.

CbNeed and the Strong Barrier. The definition of Strong CbNeed in [23] builds over the simple one in the LSC, and yet is very sophisticated and far from obvious. This is an instance of a more general fact concerning implementation techniques: dealing with the strong setting is orders of magnitude more difficult than with the closed setting, it is not just a matter of adapting a few definitions. New complex issues show up, requiring new techniques and concepts – let us refer to this fact as to *the strong barrier*. Another instance is the fact that Lévy's optimality [47] is far more complex in the strong case than in the weak one [29, 22].

For neededness, the tool to break the strong barrier is a complex notion of *needed evaluation context*, parametrized and defined by mutual induction with their sets of *needed variables*. Specifying the positions in a term where needed redexes take place is very subtle.

Reasonable Cost Models and the Strong Barrier. Another sophisticated form of sharing for λ-calculi arose recently in the study of whether the λ-calculus admits reasonable evaluation strategies, that is, strategies whose number of β steps is a reasonable time cost model (i.e. measure of time complexity) for λ-terms. The number of function calls (that is, β-steps) is the cost model often used in practice for functional programs – this is done for instance by Charguéraud and Pottier in [32]. A time cost model is *reasonable* when it is polynomially equivalent to the one of Turing machines, which is the requirement for good time cost models. For the λ-calculus, the theory justifying the practice of taking the number of function calls as a time cost model is far from trivial. It is an active research topic, see Accattoli [5].

The first result about λ-calculus reasonable strategies is due to Blelloch and Greiner [30] (1995), and concerns Closed CbV. The 2000s have seen similar results for Closed CbN and Closed CbNeed by Sands, Gustavson, and Moran [51] and Dal Lago and Martini [34, 33]. These cases are based on simulating the λ-calculus via simple forms of sharing such as those at work in abstract machines. The same kind of sharing can also be represented in the LSC, as shown by Accattoli et al. [6]. The strong case seemed elusive and was suspected not to be reasonable, because of Asperti and Mairson's result that Lévy's optimal (strong) strategy is not reasonable [21] – the elusiveness was just another instance of the strong barrier.

Useful Sharing. In 2014, Accattoli and Dal Lago managed to break the barrier, proving that Strong CbN (also known as leftmost-outermost evaluation, or *normal order*) is a reasonable strategy [14]. The proof rests on a simulation of Strong CbN in a refinement of the LSC with a new further level of sharing, deemed *useful sharing*. They also show useful sharing to be *mandatory* for breaking the strong barrier for reasonability.

Useful sharing amounts to doing minimal *unsharing* work, namely only when it contributes to creating β-steps, while avoiding to unfold the sharing (i.e. to substitute) when it only makes the term grow in size. Similarly to CbNeed, the specification of useful sharing can take place only at the micro-step level. Note that the replacement of a variable x in t with u can create a β redex only if u is (or shall reduce to) an abstraction *and* there is an applied occurrence of x in t (that is, $t = T\langle xs\rangle$ for some context T). Therefore, restricting to useful substitutions – that is, adopting useful sharing – amounts to two optimizations of the substitution/unfolding process:

1. *Never substitute normal applications*: one must avoid substitutions of terms which are not – and shall not reduce to – abstractions, such as, say, yz, because their substitution cannot create β-redexes. Indeed, $T\langle(yz)s\rangle$ has a β redex if and only if $T\langle xs\rangle$ does.
2. *Substituting abstractions on-demand*: when the term to substitute is an abstraction, one needs to be sure that the variable occurrence to replace is applied, because, for instance, replacing x with I in yx (obtaining $y\mathsf{I}$) is useless, as no β-redexes are created.

The first optimization is easy to specify, because it concerns the shape of the terms to substitute, that is, *what* to substitute – it has a small-step nature. The second one instead is very delicate, as it also concerns *where* to substitute. It depends on single variable occurrences and thus it is inherently *micro-step* – note that x has both a useful and a useless occurrence in xx. Similarly to Strong CbNeed, the difficulty is specifying *useful evaluation contexts*.

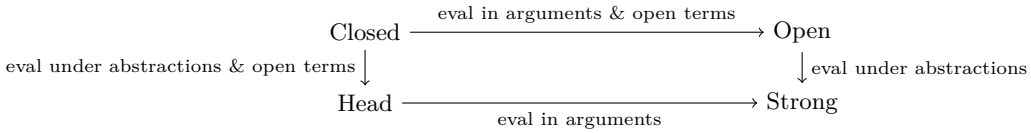
Strong CbNeed and Useful Sharing. Given the similar micro-step traits of CbNeed and useful sharing, and their similar difficulties, it is natural to wonder whether they can be combined. The operational semantics of Strong CbNeed in [23] has the easy useful optimization hardcoded, as it substitutes only abstractions. However, it ignores the delicate second optimization, and its number of β steps is therefore not a reasonable cost model. Concretely, this means that the practice of counting function calls does not reflect the cost of Balabonski et al.'s operational semantics for Strong CbNeed. Since Strong CbNeed is used in the implementation of Coq, this issue has both theoretical and practical relevance.

The aim of this paper is to start adapting useful sharing to call-by-need, developing reasonable operational semantics for CbNeed beyond the closed setting, and continuing a research line about CbNeed started by Accattoli and Barras [7, 8]. To explain our approach, we first need to overview a recent new perspective on the strong barrier.

Opening the Strong Barrier. The theory of the λ-calculus has mainly been developed in CbN. Historically, Barendregt stressed the importance of *head* evaluation (which does not evaluate arguments) for a meaningful representation of partial recursive functions – this is the leading theme of his famous book [26]. A decade later, Abramsky and Ong stressed the relevance of *weak* head evaluation (which does not evaluate abstraction bodies either) to model functional programming languages [1]. Therefore, the usual incremental way to understand strong evaluation is to start with the *closed* CbN case (i.e., weak head evaluation and closed terms), then turn to the *head* case (head evaluation and open terms), and finally add evaluation into arguments obtaining the *strong* case (and leftmost-outermost evaluation). This is for instance the progression that has been followed by Accattoli and Dal Lago to obtain a reasonable time cost model for Strong CbN [13, 14].

In a line of work by Accattoli and co-authors [9, 15, 16, 12] aimed at developing a theory of CbV beyond the usual closed case, it became clear that there is an alternative and better route to the strong setting. The idea is to consider the intermediate *open* setting (rather

than the head one) obtained by enabling evaluation in arguments and open terms (as in the strong case), while still forbidding evaluation in abstraction bodies (as in the closed case). One can summarize the situation with the following diagram:



They also show that useful sharing *factors* through the open setting, rather than through the head one: the two useful optimizations are irrelevant in the head case, while they make sense in the open one, where they can be studied without facing the whole of the strong barrier.

The strong setting can be seen as the iteration of the open one under abstraction, (but not as the iteration of the closed one, because diving into abstractions forces to deal with open terms). This view is adopted by Grégoire and Leroy in the design of the second strong abstract machine at work in Coq [38]. Useful sharing for the strong case then amounts to understanding how open useful sharing and the iteration interact, which is subtle and yet is an orthogonal problem. Studying the open case first is the progression followed recently by Accattoli and co-authors to prove that Strong CbV is reasonable for time [9, 16, 11].

This Paper. According to the decomposition of the strong barrier, here we study, as a first step, useful sharing for CbNeed in the open setting. Let us stress that, because of the barrier, it is not practicable to directly study the strong setting – this is also how the study for CbN and CbV, which are simpler than CbNeed, have been carried out in the literature.

An interesting aspect of useful sharing is that, while the underlying principle is the same, its CbN and CbV incarnations look very different, as the two strategies provide different invariants, leading to different realizations of the required optimizations. It is then interesting to explore useful sharing in CbNeed, which can be seen as a merge of CbN and CbV.

Difficulties. It turns out that useful sharing is quite more difficult to specify in CbNeed than in CbN or CbV. Useful sharing requires to know, for every variable replacement, both *what* is being substituted (is it an abstraction?) and *where* (is the variable to replace applied?). Evaluating only needed arguments, and only once, means that CbNeed evaluation moves deeply into a partially evaluated environment, making hard to keep track of both the *what* and the *where* of variable replacements. In particular, a variable might not be applied in the environment but at the same time be meant to replace an applied variable – thus being applied *up to sharing* – making the identification of applied variables a major difficulty.

The definition of useful rewriting steps is always involved. In CbN and CbV, they can nonetheless be specified compactly via the concept of *unfolding*, that is, iterated meta-level substitutions [14, 9]. These definitions can be called *semantical*, as they define useful *micro* steps via side conditions of a *small*-step nature. They are also somewhat *ineffective*, because they require further work to be made operational. Unfortunately, it is unclear how to give a semantic definition of usefulness in CbNeed. In particular, defining useful CbNeed evaluation contexts seems to require the unfolding of contexts, which is tricky, given that in CbNeed the context hole might be shared, thus risking being duplicated by the unfolding.

Outcome. Despite these difficulties, we succeed in designing an operational semantics for Open CbNeed with useful sharing, and proving that it validates the expected properties.

We proceed in three incremental steps. First, we provide a new *split* presentation of Closed CbNeed tuned for the study of useful sharing developed later on. Second, we extend it to the open setting, essentially mimicking Balabonski et al.'s approach [23], but limiting

it to the open fragment. The real novelty is the third step, providing the refinement into a *useful* open CbNeed calculus, of which we prove the good properties. The crucial and sophisticated concept is the one of *useful (CbNeed) evaluation contexts*, which isolate where useful needed substitutions can be triggered. They are parametrized and defined by mutual induction with the notions of both *applied* and *unapplied variables*, similarly to how needed evaluation contexts are parametrized and mutually dependent with needed variables. The isolation of these concepts and the proof of their properties are our main contribution.

Our definition of useful step is *operational* rather than semantical, as we give a direct – and unfortunately involved – definition of useful evaluation contexts, being unclear how to give a semantic definition based on unfoldings in CbNeed. On the positive side, ours is the first fully operational definition of usefulness in the literature. Previous work (in CbN and CbV) has either adopted semantical ones [14, 9], or has given abstract machines realizing the useful optimizations, but avoiding defining a useful calculus on purpose [3, 16, 11].

Among the properties that we prove, two can be seen as capturing the correctness and the completeness of useful sharing with respect to Open CbNeed:

- *Correctness*: useful substitution steps are eventually followed by a β step, the one that they contribute to create. That is, our useful steps correctly captures the intended semantics, as no steps irrelevant for β redexes are mistakenly considered as useful.
- *Completeness*: normal forms in Useful Open CbNeed unfold to normal forms in Open CbNeed (the unfolding of normal forms is easy to deal with). That is, useful steps do not stop *too soon*: no steps contributing to β redexes are mistakenly considered as useless.

Sketched Complexity Analysis. The third essential property for useful sharing, and its reason to be, is *reasonability*: the useful calculus can be implemented within a polynomial (or even linear) overhead in the number of β-steps. We sketch the complexity analysis at the end of the paper. A formal proof requires introducing an abstract machine implementing the calculus. We have developed the machine, but left it to a forthcoming paper for lack of space.

Intersection Types in the Background. Because of the inherent difficulties mentioned above, our calculus is involved, even very involved. To remove the suspicion that it is an ad-hoc calculus, we paired it with a characterization of its key properties via intersection types, used as a validation tool with a denotational flavor, refining the type-based studies in [41, 23, 17]. In such typing system, the delicate notions of useful evaluation contexts, and applied and unapplied variables have natural counterparts, and type derivations can be used to measure both evaluation lengths and the size of normal forms *exactly*. Such a companion study – omitted for lack of space – is in Leberle's PhD thesis [46].

Proofs. We adopt a meticulous approach, developing proofs in full details, almost at the level of a formalization in a proof assistant. The many technical details, mostly of a tedious nature, are in the technical report [18]. This paper explains the relevant concepts.

2 The Need For Useful Sharing

Here we show a paradigmatic case of size exploding family – which is a family of terms whose size grows exponentially with the number of β-steps – motivating the key optimization of useful sharing for open and strong evaluation. Actually, there are *two* paradigmatic cases of size explosion and, accordingly, *two* optimizations characterizing useful sharing. The first

optimization amounts to forbid the substitution of normal applications, and it is hardcoded into CbNeed evaluation, which by definition substitutes only values. Therefore, we omit discussing the first case of size explosion – more details can be found in [14, 11].

Size Explosion. The example of size-explosion we are concerned with is due to Accattoli [2] and based on the following families of terms, the t_i, and results, the u_i (where $\mathsf{I} := \lambda z.z$):

$$t_1 := \lambda x.\lambda y.yxx \qquad t_{n+1} := \lambda x.t_n(\lambda y.yxx) \qquad \Big| \qquad u_0 := \mathsf{I} \qquad u_{n+1} := \lambda y.yu_nu_n$$

▶ **Proposition 1** (Closed and strategy-independent size explosion, [2]). *Let $n > 0$. Then $t_n\mathsf{I} \to_\beta^n u_n$. Moreover, $|t_n\mathsf{I}| = \mathcal{O}(n)$, $|u_n| = \Omega(2^n)$, $t_n\mathsf{I}$ is closed, and u_n is normal.*

The Useful Optimization. It is easily seen that all the terms substituted along the evaluation of the family are abstractions, namely the identity I and instances of u_i, and that none of these abstractions ever becomes the abstraction (on the left) of a β-redex – that is, their substitution does not create, or it is not *useful* for, β-redexes. These abstractions are however duplicated and nested inside each other, being responsible for the exponential growth of the term size. Useful sharing is about avoiding such useless duplications.

If evaluation is weak, and substitution is *micro-step* (i.e. one variable occurrence at a time, when in evaluation position, in a formalism with sharing), then the family does not cause an explosion. The replaced variables indeed are all instances of x in some t_i which are under abstraction, and which are then never replaced in micro-step *weak* evaluation. With micro-step *strong* evaluation, however, these replacement do happen, and the size explodes. When evaluated with Balabonski et al. Strong CbNeed [23], this family takes a number of micro-steps exponential in the number of β steps, showing that – for as efficient as Strong CbNeed may be – the number of β steps does not reasonably measure its evaluation time.

To tame this problem, one needs to avoid useless substitutions, resting on an optimization sometimes called *substituting abstractions on-demand*, which is tricky. It requires abstractions to be substituted *only* on applied variable occurrences: note that the explosion is caused by replacements of variables (namely the instances of x) which are *not* applied, and that thus do not create β-redexes. For instance, the optimization should allow us substituting I on y in yx, because *it is useful*, that is, it creates a β redex, while it should forbid substituting it on x because it is *useless* for β-redexes. Note that this optimization makes sense only when one switches to micro-step evaluation, that is, at the level of machines, because in xx there are both a useful and a useless occurrence of x. The implementation of *substituting abstractions on-demand* is very subtle, also because by not performing useless substitutions, it breaks invariants of the usual open/strong evaluation process.

As shown by Accattoli and Guerrieri [16], in an open (but not strong) setting, substituting abstractions on-demand is not mandatory for reasonability. They also show, however, that it makes nonetheless sense to study it because it is mandatory for obtaining efficient implementations, as it reduces the complexity of the overhead from *quadratic* to *linear* with respect to the size of the initial term. On the other hand, the optimization is mandatory in strong settings, and it is easier to first study it in the open setting, because the iteration under abstraction (required to handle the strong case) introduces new complex subtleties.

3 The Split Presentation of Closed Call-by-Need

In this section we give an unusual *split* presentation of Closed CbNeed that shall be the starting point for our study of the open and the useful open cases of the next sections.

The Need to Split. The linear substitution calculus (LSC) provides a simple and elegant setting for studying CbNeed, as shown repeatedly by Accattoli, Kesner and co-authors [6, 41, 10, 23, 43, 17, 42]. The LSC extends the λ-calculus with *explicit substitutions* (shortened to ES), noted $t[x\leftarrow u]$, which are a compact notation for $\mathsf{let}\ x = u\ \mathsf{in}\ t$. Capture-avoiding meta-level substitution is noted $t\{x\leftarrow u\}$. To model the useful optimization explained above, we shall need to substitute abstractions only on applied variables. Now, in the LSC, ES can appear everywhere in the term, for instance there are terms such as $t := x[x\leftarrow \mathsf{I}]u$. Note that in t it is hard to say whether the replacement of x with I is useful by looking only at the scope of the ES (which is the left of the $[\cdot\leftarrow\cdot]$ construct): the subtlety being that the replacement is indeed useful, because the variable is applied and I is an abstraction, but the application it is involved in is outside the scope of the ES. To avoid this complication, we give a presentation of CbNeed where ES are separated from the term they act upon, and cannot be nested into each other, similarly to what happens in abstract machines. The split presentation is not mandatory to study useful sharing, but it is quite convenient.

Split Grammars. In the split syntax, a *term* is an ordinary λ-term (without ESs), and a *program* is a term together with – in a separate place – a list of ESs, called *environment*.

Given a countable set of variables Var, the syntax of Closed CbNeed is given by:

$$\begin{array}{llcl} \textsc{Values} & v, w & ::= & \lambda x.t \\ \textsc{Terms} & t, u, s & ::= & x \in \mathsf{Var} \mid v \mid tu \end{array} \qquad \begin{array}{llcl} \textsc{Environments} & e, e' & ::= & \epsilon \mid e[x\leftarrow t] \\ \textsc{Programs} & p, q & ::= & (t, e) \end{array}$$

Note that the body of a λ-abstraction is a term and not a program. Of course, extending the framework to strong evaluation – which is left for future work – requires to allow programs under λ-abstractions. Note also that variables are not values. This is standard in works dealing with implementations or efficiency, as excluding them brings a speed-up, as shown by Accattoli and Sacerdoti Coen [10]. In $e[x\leftarrow t]$ and $(u, e[x\leftarrow t])$ the variable x is bound in e and u. Terms and programs are identified modulo α-renaming. Environments are concatenated by simple juxtaposition. We also define the environment look-up operation as follows: set $e(x) := t$ if $e = e'[x\leftarrow t]e''$ and x is not bound in e', and $e(x) := \bot$ otherwise.

Split Contexts and Plugging. Micro-step CbN and CbV evaluation have easy *split* presentations, because their evaluation contexts may be seen as term contexts, using the environment only for look up, see for instance [8]. CbNeed evaluation contexts, instead, need to enter into the environment. Typically in a program such as $(xy, [x\leftarrow z][z\leftarrow t])$, whose head variable x has been found, CbNeed evaluation has to enter inside $[x\leftarrow\cdot]$, finding another (hereditary) head variable z, and in turn enter inside $[z\leftarrow\cdot]$ and evaluate t. The subtlety is that the evaluation of t can create new ESs, which should be added to the program without breaking its structure, that is, outside the ES which is being evaluated ($[z\leftarrow\cdot]$ in the example). The trick to make it work, is using an unusual notion of context plugging. Before defining evaluation contexts we simply discuss split contexts, which are used throughout the paper.

$$\begin{array}{llcl} \textsc{Term ctxs} & T, T' & ::= & \langle\cdot\rangle \mid Tt \mid tT \\ \textsc{Prog. ctxs} & P, Q & ::= & (T, e) \mid (t, E) \end{array} \qquad \begin{array}{llcl} \textsc{Env. ctxs} & E, E' & ::= & \epsilon \mid e\,[x\leftarrow T] \mid E\,[x\leftarrow u] \end{array}$$

Plugging of a term into a context is defined as expected. Plugging of a program into a context, the tricky bit, requires an auxiliary notation $p@[x\leftarrow t]$ for the appending of an ES $[x\leftarrow t]$ to the end of the environment of a program p:

Appending ES			Plugging of programs		
$(t,e)@[x\leftarrow u]$	$:=$	$(t,e[x\leftarrow u])$	$(T,e)\langle t,e'\rangle$	$:=$	$(T\langle t\rangle,e'e)$
$(T,e)@[x\leftarrow u]$	$:=$	$(T,e[x\leftarrow u])$	$(u,e[x\leftarrow T])\langle t,e'\rangle$	$:=$	$(u,e[x\leftarrow T\langle t\rangle]e')$
$(t,E)@[x\leftarrow u]$	$:=$	$(t,E[x\leftarrow u])$	$(u,E[x\leftarrow s])\langle t,e\rangle$	$:=$	$(u,E)\langle t,e\rangle@[x\leftarrow s]$

For instance, $(xy,[x\leftarrow t][y\leftarrow\langle\cdot\rangle][z\leftarrow u])\langle s,[x'\leftarrow t']\rangle = (xy,[x\leftarrow t][y\leftarrow s][x'\leftarrow t'][z\leftarrow u])$. The look-up operation is extended to environment and program contexts as expected.

Next, we define the CbNeed evaluation contexts in the split approach.

Head contexts	H,J	$::=$	$\langle\cdot\rangle \mid Ht$
Hereditary head contexts	H^*,J^*	$::=$	$(H,e) \mid H^*@[x\leftarrow t] \mid H^*\langle x\rangle@[x\leftarrow H]$

The third production for H^* is what allows evaluation to be iterated inside ES, seeing for instance $(xy,[x\leftarrow z][z\leftarrow\langle\cdot\rangle])$ as a hereditary head context of $(xy,[x\leftarrow z][z\leftarrow t])$ (by applying the production twice, the first time obtaining $(xy,[x\leftarrow\langle\cdot\rangle])$).

Split Evaluation Rules. In contrast to most λ-calculi, we do not define the root cases of the rules and then extend them by a closure by evaluation contexts. We rather define them directly at the global level. Adopting global rules is not mandatory, and yet it shall be convenient for dealing with the useful calculus – we use them here too for uniformity.

Closed CbNeed evaluation rules

Multiplicative	$H^*\langle(\lambda x.t)u\rangle$	$\to_{\mathsf{m}}$	$H^*\langle t,[x\leftarrow u]\rangle$	
Exponential	$H^*\langle x\rangle$	$\to_{\mathsf{e}}$	$H^*\langle v\rangle$	if $H^*(x)=v$

The names of the rules are due to the link between the LSC and linear logic, see Accattoli [4]. Note that we use both plugging of terms and programs, to ease up notations. An example of how rule $\to_{\mathsf{m}}$ exploits the unusual notion of plugging is $(xt,[x\leftarrow(\lambda y.u)st'][z\leftarrow u'])\to_{\mathsf{m}}(xt,[x\leftarrow ut'][y\leftarrow s][z\leftarrow u'])$. As it is standard in the study of CbNeed, garbage collection is simply ignored, because it is postponable at the micro-step level. Normal forms of Closed CbNeed are programs of the form (v,e), which are sometimes called *answers*.

4 Open Call-by-Need

We now shift to Open CbNeed, an evaluation strategy extending Closed CbNeed and allowing reduction to act on possibly *open* programs. Roughly, the strategy iterates CbNeed evaluation on the arguments of the head variable, when the normal form of ordinary CbNeed evaluation is not an abstraction, which can happen when terms are not necessarily closed. Various aspects of Closed CbNeed become subtler in Open CbNeed, namely the definition of evaluation contexts and the structure of normal forms, together with the new notion of needed variables. Essentially, we are giving an alternative presentation of the open fragment of Balabonski et al.'s Strong Call-by-Need, with which we compare at the end of the section.

Some Motivating Examples. We show a few examples of the rewriting relation that we aim at defining, as to guide the reader through the technical aspects. We want reduction to take place in arguments, after a (hereditary) head variable has been found, having e.g.:

$$(x((\lambda z.z)\mathsf{I}),[y\leftarrow t])\to_{\mathsf{m}}(xz,[z\leftarrow\mathsf{I}][y\leftarrow t]) \quad\text{and}\quad (xz,[z\leftarrow\mathsf{I}][y\leftarrow t])\to_{\mathsf{e}}(x\mathsf{I},[z\leftarrow\mathsf{I}][y\leftarrow t])$$

For appropriate generalizations of $\to_{\mathsf{m}}$ and $\to_{\mathsf{e}}$. Of course, we retain and extend to arguments the hereditary character of the reduction rules, therefore having also steps such as:

$$(yx, [x{\leftarrow}y((\lambda z.z)\mathsf{I})]) \to_{\mathsf{m}} (yx, [x{\leftarrow}yz][z{\leftarrow}\mathsf{I}]), \text{ and } (yx, [x{\leftarrow}yz][z{\leftarrow}\mathsf{I}]) \to_{\mathsf{e}} (yx, [x{\leftarrow}y\mathsf{I}][z{\leftarrow}\mathsf{I}])$$

While the intended behavior is – we hope – clear, specifying these steps via evaluation contexts requires some care and a few definitions. Essentially, we need to understand when evaluation can pass to the next argument, and thus characterize when terms are normal. This is easy for terms but becomes tricky for programs.

Evaluation Places and Needed Variables. The grammars of the language are the same as for split Closed CbNeed, but defining the open evaluation contexts is quite subtler. In Closed CbNeed there is only one place of the term where evaluation can take place, the hereditary head context H^*. In the open setting the situation is more general: there is one *active* evaluation place *plus* potentially many *passive* ones, which are those places where evaluation already passed and ended. On some of these passive places, evaluation ended on a free variable (occurrence). We refer to these free variables as *needed* (definition below[1]), as they shall end up in the normal form, given that at least one of their occurrences has already been evaluated and cannot be erased. For instance in $p := (x(y\mathsf{I}), [z{\leftarrow}x][y{\leftarrow}\mathsf{I}])$ the active place is y, the first occurrence of x is a needed occurrence, while the second one is not.

Needed vars for terms

$$\begin{aligned} \mathsf{nv}(x) &:= \{x\} \\ \mathsf{nv}(\lambda x.t) &:= \emptyset \\ \mathsf{nv}(tu) &:= \mathsf{nv}(t) \cup \mathsf{nv}(u) \end{aligned}$$

Needed variables for programs

$$\begin{aligned} \mathsf{nv}(t, \epsilon) &:= \mathsf{nv}(t) \\ \mathsf{nv}(t, e[x{\leftarrow}u]) &:= \begin{cases} \mathsf{nv}(t,e) & x \notin \mathsf{nv}(t,e) \\ (\mathsf{nv}(t,e) \setminus \{x\}) \cup \mathsf{nv}(u) & x \in \mathsf{nv}(t,e) \end{cases} \end{aligned}$$

The difficulty in defining Open CbNeed is in the inductive definition of both normal forms and evaluation contexts. The problem is that extending a term or a context with a new ES may re-activate a passive evaluation place, if the ES binds a needed variable occurrence. For instance, appending $[x{\leftarrow}\delta]$ to p above would reactivate the needed occurrence of x.

Normal Terms. In Open CbNeed normal forms are not simply answers (i.e. abstractions together with an environment), as free variables induce a richer structure. We shall later characterize the subtle inductive structure of normal programs. For now, we need predicates (that shall be later shown) characterizing normal *terms*, as they are used to define evaluation contexts. The definition and the terminology are borrowed from Open CbV [9, 15], where normal terms are called *fireballs* and are defined by mutual induction with *inert terms*:

$$\begin{array}{rrcl} \text{Values} & v, w & ::= & \lambda x.t \\ \text{Fireballs} & f, g & ::= & v \mid i \end{array} \qquad \begin{array}{rrcl} \text{Inert terms} & i, j & ::= & x \in \mathsf{Var} \mid if \\ \text{Non-var inert terms} & i^+ & ::= & if \end{array}$$

Later on, we shall often need to refer to inert terms that are not variables, which is why we introduce now a dedicated notation. We shall sometimes write $\mathsf{inert}(t)$ (resp., $\mathsf{abs}(t)$) to express that t is an inert term (resp., an abstraction).

[1] Needed variables are intended to be considered only for normal terms (or normal programs, or normal parts of a context), and yet the definition is given here for every term (in particular every applications, instead of only inert applications if). The reason for our lax definition is that the technical development requires at times to consider the needed variables of a term that is not yet known to be normal. The lax definition goes against the *needed* intuition, as one of the reviewers understandably complained about, suggesting to call these variables *frozen*, following Balabonski et al. [23]. We preferred to keep *needed* because they are similar but different from the frozen variables in [23], see the end of this section.

Needed vars for term ctxs

$$\mathsf{nv}(\langle\cdot\rangle) \coloneqq \emptyset \qquad \mathsf{nv}(Ht) \coloneqq \mathsf{nv}(H) \qquad \mathsf{nv}(iH) \coloneqq \mathsf{nv}(i) \cup \mathsf{nv}(H)$$

Open evaluation contexts and their needed vars

$$\frac{}{(H,\epsilon) \in \mathcal{E}_{\mathsf{nv}(H)}}\ \mathsf{O}_{\mathsf{AX}} \qquad \frac{P \in \mathcal{E}_{\mathcal{V}} \quad x \in \mathcal{V}}{P@[x{\leftarrow}i] \in \mathcal{E}_{(\mathcal{V}\setminus\{x\})\cup\mathsf{nv}(i)}}\ \mathsf{O}_{\mathsf{I}}$$

$$\frac{P \in \mathcal{E}_{\mathcal{V}} \quad x \notin \mathcal{V}}{P@[x{\leftarrow}t] \in \mathcal{E}_{\mathcal{V}}}\ \mathsf{O}_{\mathsf{GC}} \qquad \frac{P \in \mathcal{E}_{\mathcal{V}} \quad x \notin \mathcal{V}}{P\langle x\rangle@[x{\leftarrow}H] \in \mathcal{E}_{\mathcal{V}\cup\mathsf{nv}(H)}}\ \mathsf{O}_{\mathsf{HER}}$$

Figure 1 Needed variables for term contexts and the derivation rules for open evaluation contexts.

$$\frac{}{\mathsf{inert}(i,\epsilon)}\ \mathsf{I}_{\mathsf{AX}} \qquad \frac{\mathsf{inert}(p) \quad x \in \mathsf{nv}(p)}{\mathsf{inert}(p@[x{\leftarrow}i])}\ \mathsf{I}_{\mathsf{I}} \qquad \frac{\mathsf{inert}(p) \quad x \notin \mathsf{nv}(p)}{\mathsf{inert}(p@[x{\leftarrow}t])}\ \mathsf{I}_{\mathsf{GC}}$$

$$\frac{}{\mathsf{abs}(v,\epsilon)}\ \mathsf{A}_{\mathsf{AX}} \qquad \frac{\mathsf{abs}(p)}{\mathsf{abs}(p@[x{\leftarrow}t])}\ \mathsf{A}_{\mathsf{GC}}$$

Figure 2 Predicates for Open CbNeed normal programs.

Evaluation Contexts. Open evaluation contexts cannot be defined with a grammar, as for the closed case, because they are defined by mutual induction with their own set of needed variables, see the right part of Fig. 1. The notation $P \in \mathcal{E}_{\mathcal{V}}$ means that P is an open evaluation context of needed variables $\mathcal{V}$. We assume that $x \notin \mathsf{dom}(P)$ in rules O_{GC}, O_{I} and $\mathsf{O}_{\mathsf{HER}}$, in accordance with *Barendregt's variable convention*. The base case O_{AX} requires the notion of needed variables for term contexts, which is on the left side of Fig. 1.

Rule O_{AX} simply coerces term contexts to program contexts. The production $P@[x{\leftarrow}t]$ for the closed case here splits into the two rules O_{GC} and O_{I}. This is relative to needed variables: one can append the ES $[x{\leftarrow}t]$ only if x is not needed (O_{GC}) or, when x is needed, if the content t of the ES is inert (O_{I}), as to avoid re-activation of a passive evaluation place on x. Rule $\mathsf{O}_{\mathsf{HER}}$ is the open version of the production $P\langle x\rangle@[x{\leftarrow}H]$, with the needed variables constraint to prevent re-activations. Examples: $(xy, [y{\leftarrow}\langle\cdot\rangle])$ and $(xy, [y{\leftarrow}z][z{\leftarrow}\langle\cdot\rangle])$ for $\mathsf{O}_{\mathsf{HER}}$, $(xy, [y{\leftarrow}\langle\cdot\rangle][x{\leftarrow}zz])$ for O_{I}, $(xy, [y{\leftarrow}\langle\cdot\rangle][z{\leftarrow}zz])$ for O_{I}.

▶ **Lemma 2** (Unique parameterization of open evaluation contexts)**.** *Let $P \in \mathcal{E}_{\mathcal{V}}$ and $P \in \mathcal{E}_{\mathcal{W}}$. Then $\mathcal{V} = \mathcal{W}$.*

Open Evaluation Rules. The definition of the evaluation rules mimics exactly the one for the split closed case. Given an Open CbNeed evaluation context $P \in \mathcal{E}_{\mathcal{V}}$, we have:

Open CbNeed evaluation rules

$$\begin{array}{rlll} \text{Open multiplicative} & P\langle(\lambda x.t)u\rangle & \to_{\mathsf{om}} & P\langle t, [x{\leftarrow}u]\rangle \\ \text{Open exponential} & P\langle x\rangle & \to_{\mathsf{oe}} & P\langle v\rangle \qquad \text{if } P(x) = v \end{array}$$

We shall say that p reduces to q in the Open CbNeed evaluation strategy, and write $p \to_{\mathsf{ond}} q$, whenever $p \to_{\mathsf{om}} q$ or $p \to_{\mathsf{oe}} q$.

▶ **Proposition 3** (Determinism of Open CbNeed)**.** *Reduction $\to_{\mathsf{ond}}$ is deterministic.*

Normal Programs. Normal programs mimic normal terms and are of two kinds, inert or abstractions. The definition however now depends on needed variables and cannot be given as a simple grammar. The two predicates inert and abs are defined in Fig. 2. Finally, predicate onorm is defined as the union of inert and abs, that is, $\mathsf{onorm}(p)$ if $\mathsf{inert}(p)$ or $\mathsf{abs}(p)$. The intended meaning is that it characterizes programs in Open CbNeed-normal form.

▶ **Proposition 4** (Syntactic characterization of Open CbNeed-normal forms). *Let p be a program. Then p is in $\to_{\mathsf{ond}}$-normal form if and only if* $\mathsf{onorm}(p)$.

The proofs of Prop. 3 and 4 (in [18]) are subtler and longer than one might expect, because of the fact that evaluation contexts and needed variables are mutually defined.

Relationship with Balabonski et al. With respect to the definition of Strong CbNeed in [23], we follow essentially the same approach up to two differences, not counting the obvious fact that we are open and not strong. First, we use a split calculus, while they do not, because they do not study useful sharing.

Second, they have a similar but different parametrization of evaluation contexts. They are more liberal, as their sets of *frozen variables* used as parameters are supersets of our needed variables, but they also parametrize reduction steps, which we avoid. Our 'tighter' choice is related to the fine study of intersection types for Open CbNeed, which can be found in the Leberle's PhD thesis [46], and it is also essential for the refinement required by the useful extension of Sect. 5. In [24], a reformulation of [23] using a deductive system (parametrized by frozen variables) rather than evaluation contexts is used – it could also be used here.

5 Useful Open Call-by-Need

Roughly, useful sharing is an optimization of micro-step substitutions, that is, of exponential steps. The idea is that there are substitution steps that are useful to create β/multiplicative redexes and steps that are useless. For instance (the underline stresses the created β-redex):

Example of useful step	Example of useless step
$(xy, [x\leftarrow \mathsf{I}]) \to_{\mathsf{oe}} (\underline{\mathsf{I}y}, [x\leftarrow \mathsf{I}])$	$(xy, [y\leftarrow \mathsf{I}]) \to_{\mathsf{oe}} (x\mathsf{I}, [y\leftarrow \mathsf{I}])$

The main idea is that useful steps replace applied variable occurrences, while useless steps replace unapplied occurrences. The definition of the useful calculus then shall refine the open one by replacing the set of needed variables with *two* sets, one for applied and one for unapplied variable occurrences. Note a subtlety: variables can have both applied and unapplied needed occurrences, as x in xx. Therefore, usefulness is a concept that can be properly expressed only when considering replacements of single variable occurrences.

Usefulness unfortunately is not so simple. Consider the following step replacing z with I:

$$(xy, [x\leftarrow z][z\leftarrow \mathsf{I}]) \to_{\mathsf{oe}} (xy, [x\leftarrow \mathsf{I}][z\leftarrow \mathsf{I}]) \tag{1}$$

Is it useful or useless? It does not create a multiplicative redex – therefore it looks useless – but without it we cannot perform the next step $(xy, [x\leftarrow \mathsf{I}][z\leftarrow \mathsf{I}]) \to_{\mathsf{oe}} (\mathsf{I}y, [x\leftarrow \mathsf{I}][z\leftarrow \mathsf{I}])$ replacing x with I which is certainly useful – thus step (1) *has to be useful*.

We then have to refine the defining principle for usefulness: useful steps replace *hereditarily applied* variable occurrences, that is, occurrences that are applied, or that are by themselves (i.e. not in an application) and that are meant to replace a hereditarily applied occurrence.

Handling hereditarily applied variables is specific to CbNeed, and makes defining Useful Open CbNeed quite painful. The key point is the *global* character of the hereditary notion, that requires checking the evaluation context leading to the variable occurrence and it is then not of a local nature. We believe that hereditarily applied variables, nonetheless, are an unavoidable ingredient of usefulness in a CbNeed scenario, and not an ad-hoc point of our study. This opinion is backed by the fact that such a convoluted mechanism is modeled very naturally at the level of intersection types, as it is shown in Leberle's PhD Thesis [46].
Important: from now on, we ease the language saying *applied* to mean *hereditarily applied*.

Applied variables for terms and programs

$$\mathsf{a}(\lambda x.t) \coloneqq \emptyset \qquad \mathsf{a}(x) \coloneqq \emptyset \qquad \mathsf{a}(tu) \coloneqq \begin{cases} \{x\} \cup \mathsf{a}(u) & t = x \in \mathsf{Var} \\ \mathsf{a}(t) \cup \mathsf{a}(u) & t \notin \mathsf{Var} \end{cases} \qquad \mathsf{a}(t,\epsilon) \coloneqq \mathsf{a}(t)$$

$$\mathsf{a}(t, e[x{\leftarrow}u]) \coloneqq \begin{cases} \mathsf{a}(t,e) & x \notin \mathsf{nv}(t,e), \\ (\mathsf{a}(t,e){\setminus}\{x\}) \cup \mathsf{a}(u) & x \in \mathsf{nv}(t,e) \wedge (x \notin \mathsf{a}(t,e) \vee u \notin \mathsf{Var}), \\ (\mathsf{a}(t,e){\setminus}\{x\}) \cup \{y\} & x \in \mathsf{nv}(t,e) \wedge x \in \mathsf{a}(t,e) \wedge u = y \in \mathsf{Var} \end{cases}$$

Unapplied variables for terms and programs

$$\mathsf{u}(\lambda x.t) \coloneqq \emptyset \qquad \mathsf{u}(x) \coloneqq \{x\} \qquad \mathsf{u}(tu) \coloneqq \begin{cases} \mathsf{u}(u) & t \in \mathsf{Var} \\ \mathsf{u}(t) \cup \mathsf{u}(u) & t \notin \mathsf{Var} \end{cases} \qquad \mathsf{u}(t,\epsilon) \coloneqq \mathsf{u}(t)$$

$$\mathsf{u}(t, e[x{\leftarrow}u]) \coloneqq \begin{cases} \mathsf{u}(t,e) & x \notin \mathsf{u}(t,e) \wedge (x \notin \mathsf{nv}(t,e) \vee u = y \in \mathsf{Var}) \\ (\mathsf{u}(t,e){\setminus}\{x\}) \cup \mathsf{u}(u) & x \in \mathsf{u}(t,e) \vee (x \in \mathsf{nv}(t,e) \wedge u \notin \mathsf{Var}) \end{cases}$$

Applied vars of term contexts

$$\begin{array}{rcl} \mathsf{a}(\langle\cdot\rangle) & \coloneqq & \emptyset \\ \mathsf{a}(Ht) & \coloneqq & \mathsf{a}(H) \\ \mathsf{a}(iH) & \coloneqq & \mathsf{a}(i) \cup \mathsf{a}(H) \end{array}$$

Unapplied vars of term contexts

$$\begin{array}{rcl} \mathsf{u}(\langle\cdot\rangle) & \coloneqq & \emptyset \\ \mathsf{u}(Ht) & \coloneqq & \mathsf{u}(H) \\ \mathsf{u}(iH) & \coloneqq & \mathsf{u}(i) \cup \mathsf{u}(H) \end{array}$$

Figure 3 Applied and unapplied variables for terms, programs, and term contexts.

Applied and Unapplied Variables. We now define, for terms, programs, and term contexts, the sets of applied and unapplied variables $\mathsf{a}(\cdot)$ and $\mathsf{u}(\cdot)$, that are subsets of needed variables $\mathsf{nv}(\cdot)$. We shall prove that $\mathsf{nv}(t) = \mathsf{a}(t) \cup \mathsf{u}(t)$ (i.e., the two sets cover $\mathsf{nv}(t)$ exactly). As already pointed out, applied and unapplied variables, however, are *not* a partition of needed variables, that is, in general $\mathsf{a}(t) \cap \mathsf{u}(t) \neq \emptyset$ as a variable can have both applied and unapplied (needed) occurrences, as x in xx. The same holds also for programs and term contexts.

The set of applied variables of terms, programs, and term contexts are defined in Fig. 3 – explanations follow. Having in mind that we want to define $\mathsf{a}(p)$ in such a way that it satisfies $\mathsf{a}(p) \subseteq \mathsf{nv}(p)$, note that condition $x \in \mathsf{nv}(t,e) \wedge x \in \mathsf{a}(t,e) \wedge u = y \in \mathsf{Var}$ in the definition of $\mathsf{a}(t, e[x{\leftarrow}u])$ would more simply be $x \in \mathsf{a}(t,e) \wedge u = y \in \mathsf{Var}$. However, we have not proved yet that $\mathsf{a}(p) \subseteq \mathsf{nv}(p)$, which is why the definition is given in this more general form.

We give some examples. As expected, y is an applied variable of $y\,z$ and $z\,(y\,z)$. It is also applied in $p \coloneqq (z\,x, [x{\leftarrow}y\,z])$, even if x is not applied in $(z\,x, \epsilon)$. Thus, Useful Open CbNeed evaluation shall be defined as to include exponential steps such as $(z\,x, [x{\leftarrow}y\,z][z{\leftarrow}v]) \to_{\mathsf{oe}} (z\,x, [x{\leftarrow}v\,z][y{\leftarrow}v])$, which are useful. Note that y is not applied in $(x, [z{\leftarrow}yx])$, because applied variables have to be needed variables, and y is not needed. Another example: if $p \coloneqq (xt, [x{\leftarrow}y])$, then $y \in \mathsf{a}(p)$ (and also $z \in \mathsf{a}((xt, [x{\leftarrow}y][y{\leftarrow}z]))$). Useful Open CbNeed, then, shall retain the following two exponential steps of the open case, since the sequence is supposed to continue with a $\to_{\mathsf{m}}$ step, contracting the redex given by vt:

$$(xt, [x{\leftarrow}y][y{\leftarrow}v]) \to_{\mathsf{oe}} (xt, [x{\leftarrow}v][y{\leftarrow}v]) \to_{\mathsf{oe}} (vt, [x{\leftarrow}v][y{\leftarrow}v])$$

The set of unapplied variables of terms, programs, and term contexts are defined in Fig. 3. Once again, in the second clause defining $\mathsf{u}(t, e[x{\leftarrow}u])$ the side condition $x \in \mathsf{nv}(t,e)$ can be replaced by $x \in \mathsf{a}(t,e)$, after Lemma 5 below is proved.

$$\frac{}{(H,\epsilon) \in \mathcal{E}_{\mathsf{u}(H),\mathsf{a}(H)}}\ \mathsf{M}_{\mathsf{AX}} \qquad \frac{P \in \mathcal{E}_{\mathcal{U},\mathcal{A}} \quad x \in (\mathcal{U} \cup \mathcal{A})}{P@[x{\leftarrow}y] \in \mathcal{E}_{\mathsf{upd}(\mathcal{U},x,y),\mathsf{upd}(\mathcal{A},x,y)}}\ \mathsf{M}_{\mathsf{VAR}}$$

$$\frac{P \in \mathcal{E}_{\mathcal{U},\mathcal{A}} \quad x \notin (\mathcal{U} \cup \mathcal{A})}{P@[x{\leftarrow}t] \in \mathcal{E}_{\mathcal{U},\mathcal{A}}}\ \mathsf{M}_{\mathsf{GC}} \qquad \frac{P \in \mathcal{E}_{\mathcal{U},\mathcal{A}} \quad x \in (\mathcal{U} \cup \mathcal{A})}{P@[x{\leftarrow}i^+] \in \mathcal{E}_{(\mathcal{U}\setminus\{x\})\cup\mathsf{u}(i^+),(\mathcal{A}\setminus\{x\})\cup\mathsf{a}(i^+)}}\ \mathsf{M}_{\mathsf{I}}$$

$$\frac{P \in \mathcal{E}_{\mathcal{U},\mathcal{A}} \quad x \in (\mathcal{U} \setminus \mathcal{A})}{P@[x{\leftarrow}v] \in \mathcal{E}_{\mathcal{U}\setminus\{x\},\mathcal{A}}}\ \mathsf{M}_{\mathsf{U}} \qquad \frac{P \in \mathcal{E}_{\mathcal{U},\mathcal{A}} \quad x \notin (\mathcal{U} \cup \mathcal{A})}{P\langle x\rangle@[x{\leftarrow}H] \in \mathcal{E}_{\mathcal{U}\cup\mathsf{u}(H),\mathcal{A}\cup\mathsf{a}(H)}}\ \mathsf{M}_{\mathsf{HER}}$$

Figure 4 Derivation rules for multiplicative evaluation contexts.

We give some examples. A consequence of the definition is that, as for applied variables, y is not unapplied in $(xx, [z{\leftarrow}xy])$ because it is not needed. As it is probably expected, y is unapplied in $(zx, [z{\leftarrow}xy])$, even if xy is meant to replace z which is applied in zx. Perhaps counter-intuitively, instead, our definitions imply both $y \in \mathsf{a}(p)$ and $y \in \mathsf{u}(p)$ for $p := (xx, [x{\leftarrow}y])$, that is, the unique occurrence of y is both applied and unapplied in p[2].

▶ **Lemma 5** (Unapplied and applied cover needed variables)**.**

1. Terms*:* $\mathsf{nv}(t) = \mathsf{u}(t) \cup \mathsf{a}(t)$ *for every term* t*.*
2. Programs*:* $\mathsf{nv}(p) = \mathsf{u}(p) \cup \mathsf{a}(p)$ *for every program* p*.*
3. Term contexts*:* $\mathsf{nv}(H) = \mathsf{u}(H) \cup \mathsf{a}(H)$*, for every term context* H*.*

Finally, the derived concept of useless variable shall also be used.

▶ **Definition 6** (Useless variables)**.** *Given a term* t*, we define the set of* useless *variables as* $\mathsf{ul}(t) := \mathsf{u}(t) \setminus \mathsf{a}(t)$*. The set of* useless *variables of a program* p *is defined analogously.*

Useless variables are crucial in differentiating Useful Open CbNeed from Open CbNeed. We shall prove that if p is a useful open normal form and $x \in \mathsf{ul}(p)$, then $p@[x{\leftarrow}v]$ is also a useful open normal form (while it is not a open normal form). The notion of useless variables is intuitively simple but technically complex. Some examples. First, note that $\mathsf{ul}(x\,x, \epsilon) = \emptyset$. The example can be extended to a hereditary setting, noting that $\mathsf{ul}(y, [y{\leftarrow}x\,x]) = \emptyset$. However, the reasoning takes into account only needed occurrences, that is, note that $x \in \mathsf{ul}(z\,x, [y{\leftarrow}x\,x])$, as the occurrence of x that is applied to an argument is not needed.

Evaluation Contexts. The definition of evaluation contexts is particularly subtle in the useful case. First of all, their set $\mathcal{E}_{\mathcal{U},\mathcal{A}}$ is indexed by *two* sets of variables (rather than one as in the open case), the applied $\mathcal{A}$ and the unapplied $\mathcal{U}$ variables of the context, defined by mutual induction with the contexts themselves. The second key point is that there are two different kinds of evaluation contexts, a permissive one for multiplicative redexes, whose set is noted $\mathcal{E}_{\mathcal{U},\mathcal{A}}$, and a restrictive one for exponential redexes, noted $\mathcal{E}^{@}_{\mathcal{U},\mathcal{A}}$ and implementing the fact that the variable occurrence to be replaced has to be in an applied position. The asymmetry is unavoidable, because useful sharing concerns only exponential steps.

[2] This fact is in accordance with the companion intersection type study in Leberle's PhD thesis [46] mentioned in the introduction: in spite of y having only one syntactic occurrence in p, it is needed twice, and so intersection types derivations of p do type y twice.

$$\frac{}{(H^{@},\epsilon)\in\mathcal{E}^{@}_{\mathsf{u}(H^{@}),\mathsf{a}(H^{@})}}\ \mathsf{E}_{\mathsf{AX}_1} \qquad \frac{P\in\mathcal{E}^{@}_{\mathcal{U},\mathcal{A}}\quad x\notin(\mathcal{U}\cup\mathcal{A})}{P\langle x\rangle@[x{\leftarrow}H^{@}]\in\mathcal{E}^{@}_{(\mathcal{U}\setminus\{x\})\cup\mathsf{u}(H^{@}),\mathcal{A}\cup\mathsf{a}(H^{@})}}\ \mathsf{E}_{\mathsf{AX}_2}$$

$$\frac{P\in\mathcal{E}^{@}_{\mathcal{U},\mathcal{A}}\quad x\in(\mathcal{U}\cup\mathcal{A})}{P@[x{\leftarrow}y]\in\mathcal{E}^{@}_{\mathsf{upd}(\mathcal{U},x,y),\mathsf{upd}(\mathcal{A},x,y)}}\ \mathsf{E}_{\mathsf{VAR}} \qquad \frac{P\in\mathcal{E}^{@}_{\mathcal{U},\mathcal{A}}\quad x\in(\mathcal{U}\cup\mathcal{A})}{P@[x{\leftarrow}i^{+}]\in\mathcal{E}^{@}_{(\mathcal{U}\setminus\{x\})\cup\mathsf{u}(i^{+}),(\mathcal{A}\setminus\{x\})\cup\mathsf{a}(i^{+})}}\ \mathsf{E}_{\mathsf{I}}$$

$$\frac{P\in\mathcal{E}^{@}_{\mathcal{U},\mathcal{A}}\quad x\notin(\mathcal{U}\cup\mathcal{A})}{P@[x{\leftarrow}t]\in\mathcal{E}^{@}_{\mathcal{U},\mathcal{A}}}\ \mathsf{E}_{\mathsf{GC}} \qquad \frac{P\in\mathcal{E}^{@}_{\mathcal{U},\mathcal{A}}\quad x\in(\mathcal{U}\setminus\mathcal{A})}{P@[x{\leftarrow}v]\in\mathcal{E}^{@}_{\mathcal{U}\setminus\{x\},\mathcal{A}}}\ \mathsf{E}_{\mathsf{U}} \qquad \frac{P\in\mathcal{E}^{@}_{\mathcal{U},\mathcal{A}}\quad x\notin\mathcal{A}}{P\langle x\rangle@[x{\leftarrow}\langle\cdot\rangle]\in\mathcal{E}^{@}_{\mathcal{U}\setminus\{x\},\mathcal{A}}}\ \mathsf{E}_{\mathsf{NA}}$$

Figure 5 Derivation rules for exponential evaluation contexts.

Multiplicative Contexts. They are a refinement of the open contexts defined in Fig. 4. Their set is noted $\mathcal{E}_{\mathcal{U},\mathcal{A}}$. The refinement is needed even if useful sharing concerns only exponential steps: a multiplicative context such as $((yx)\langle\cdot\rangle,[x{\leftarrow}v])\in\mathcal{E}_{\emptyset,\{y\}}$ indeed is not an open context, because it contains a useless substitution step that in Open CbNeed would be fired before evaluating the hole. The definition of multiplicative contexts follows the one for Open CbNeed contexts (M_{AX}, M_{GC}, and $\mathsf{M}_{\mathsf{HER}}$ are essentially as before) *except for* rule:

$$\frac{P\in\mathcal{E}_{\mathcal{V}}\quad x\in\mathcal{V}}{P@[x{\leftarrow}i]\in\mathcal{E}_{(\mathcal{V}\setminus\{x\})\cup\mathsf{nv}(i)}}\ \mathsf{O}_{\mathsf{I}}$$

which is now generalized into 3 rules, depending on the kind of term contained in the ES. That is, given $P\in\mathcal{E}_{\mathcal{U},\mathcal{A}}$ and $x\in(\mathcal{U}\cup\mathcal{A})$, the constraints to extend P with an ES $[x{\leftarrow}t]$ are:

- *Rule* M_{I}: there are no constraints if t is a non-variable inert term i^{+}. Note that M_{I} and M_{GC} together imply that we can *always* append ESs containing inert terms to multiplicative contexts, without altering the Useful Open CbNeed order of reduction.
- *Rule* $\mathsf{M}_{\mathsf{VAR}}$: this rule covers the case where t is a variable y. It is used to handle the global applicative constraint, as in such a case, if the evaluation context is $P@[x{\leftarrow}y]$, then y has to be added to the applied and/or unapplied variables of the context, according to the role played by x in P, which is realized via the function upd defined as follows:

$$\mathsf{upd}(S,x,y):=\begin{cases}S & x\notin S\\ (S\setminus\{x\})\cup\{y\} & x\in S\end{cases}$$

- *Rule* M_{U}: it covers the case where t is a value v, requiring that x is not applied, that is, $\notin\mathcal{A}$. Such an extension would have re-activated x in the plain open case, and created a (useless) exponential redex, but here it shall not be the case. Note that it means that $P@[x{\leftarrow}t]$ is a multiplicative context *only* if $x\in(\mathcal{U}\setminus\mathcal{A})$, i.e. if x is a *useless* variable of P.

Exponential Contexts. Exponential contexts are even more involved, because they have to select only *applicative* variable occurrences and the applicative constraint is of a global nature. First, we need a notion of applicative term context, where the hole is applied.

▶ **Definition 7** (Applicative term contexts). *A term context H shall be called an* applicative term context *if it is derived using the grammar $H^{@},J^{@},I^{@}::=\langle\cdot\rangle t\mid H^{@}t\mid iH^{@}$.*

▶ **Definition 8** (Exponential evaluation contexts). *We shall say that an evaluation context P is a* exponential evaluation context *if it is derived with the rules in Fig. 5.*

Applicative term contexts serve as the base case of exponential evaluation contexts, now given by two refinements of the multiplicative case:

1. the base case $\mathsf{E}_{\mathsf{AX}_1}$ is akin to the base case M_{AX} for multiplicative contexts, except that it requires the term context to be applicative.
2. the plugging-based rule $\mathsf{M}_{\mathsf{HER}}$ splits in two. A first rule $\mathsf{E}_{\mathsf{AX}_2}$ which simply plugs an applicative context $H^{@}$ into a *multiplicative* evaluation context – note that this rule gives another base case for exponential evaluation contexts. A second rule E_{NA} that handles the special case of the global applicative constraint.

Let us see the differences between rules E_{NA} and $\mathsf{E}_{\mathsf{AX}_2}$ with two examples. Their side conditions ($x \notin (\mathcal{U} \cup \mathcal{A})$ and $x \notin \mathcal{A}$) are explained at page 17 of the technical report [18].

- E_{NA}: consider the program $p := (x\,t, [x{\leftarrow}z])$, where z is in applied position due to the global applicative constraint, as it substitutes x which is applied to t. We may derive an exponential evaluation context P that isolates z, that is, such that $P\langle z\rangle = p$, as follows:

$$\dfrac{\dfrac{}{(\langle\cdot\rangle\, t, \epsilon) \in \mathcal{E}^{@}_{\emptyset,\emptyset}}\ \mathsf{E}_{\mathsf{AX}_1} \quad x \notin \emptyset}{P := ((\langle\cdot\rangle\, t, \epsilon)\langle x\rangle)\,@[x{\leftarrow}\langle\cdot\rangle] \in \mathcal{E}^{@}_{\emptyset,\emptyset}}\ \mathsf{E}_{\mathsf{NA}}$$

  noting that $P = ((\langle\cdot\rangle\, t, \epsilon)\langle x\rangle)\,@[x{\leftarrow}\langle\cdot\rangle] = (x\,t, [x{\leftarrow}\langle\cdot\rangle])$, and so $p = P\langle z\rangle$ as expected. In this case, we are extending an exponential context, which is already applied.
- $\mathsf{E}_{\mathsf{AX}_2}$: consider $p := (x, [x{\leftarrow}z\,t])$, for which z is an applied variable because it is itself applied, while its ES binds the needed but unapplied variable x. Let us derive an exponential evaluation context P focusing on z in such a way that $P\langle z\rangle = p$ as follows:

$$\dfrac{\dfrac{}{(\langle\cdot\rangle, \epsilon) \in \mathcal{E}_{\emptyset,\emptyset}}\ \mathsf{O}_{\mathsf{AX}} \quad x \notin (\emptyset \cup \emptyset)}{P := ((\langle\cdot\rangle, \epsilon))\,\langle x\rangle @[x{\leftarrow}\langle\cdot\rangle\, t] \in \mathcal{E}^{@}_{\emptyset,\emptyset}}\ \mathsf{E}_{\mathsf{AX}_2}$$

  noting that $P = ((\langle\cdot\rangle, \epsilon))\,\langle x\rangle @[x{\leftarrow}\langle\cdot\rangle\, t] = (x, [x{\leftarrow}\langle\cdot\rangle\, t])$, and so $p = P\langle z\rangle$ as expected. Here the context $(\langle\cdot\rangle, \epsilon)$ in the hypothesis is *multiplicative* and it becomes *exponential* once extended with an ES containing an applicative term context.

The next proposition guarantees that exponential contexts are a restriction of multiplicative contexts, that is, that the introduced variations over the deduction rules do not add contexts that were not already available before.

▶ **Proposition 9** (Exponential contexts are multiplicative). *Let $P \in \mathcal{E}^{@}_{\mathcal{U},\mathcal{A}}$. Then $P \in \mathcal{E}_{\mathcal{V},\mathcal{B}}$, for some $\mathcal{V} \subseteq \mathcal{U}$ and $\mathcal{B} \subseteq \mathcal{A}$.*

Let us repeat that, instead, multiplicative contexts are not in general exponential contexts, because they are not required to be applicative, for instance $P := (x\langle\cdot\rangle, [x{\leftarrow}yy]) \in \mathcal{E}_{\{y\},\{y\}}$ is a multiplicative context but not an exponential one.

Evaluation Rules. The reduction rules for the Useful Open CbNeed strategy are:

Useful Open CbNeed evaluation rules		
Useful multiplicative	$P\langle(\lambda x.t)u\rangle \to_{\mathsf{um}} P\langle t, [x{\leftarrow}u]\rangle$	if $P \in \mathcal{E}_{\mathcal{U},\mathcal{A}}$
Useful exponential	$P\langle x\rangle \to_{\mathsf{ue}} P\langle v\rangle$	if $P \in \mathcal{E}^{@}_{\mathcal{U},\mathcal{A}}$ and $P(x) = v$

Moreover, we shall say that p reduces in the Useful Open CbNeed strategy to q, and write $p \to_{\mathsf{und}} q$, if $p \to_{\mathsf{um}} q$ or $p \to_{\mathsf{ue}} q$.

Determinism. The first property of useful evaluation that we consider is determinism, that is proved similarly for the open case, but for some further technicalities due to the existence of two sets of variables parametrizing evaluation contexts.

▶ **Proposition 10** (Determinism of Useful Open CbNeed). $\to_{\mathsf{und}}$ *is deterministic.*

Usefulness. We prove two properties ensuring that the defined reduction captures useful sharing. The first one is a *correctness* property, stating that useful exponential steps are eventually followed by a multiplicative step – *no useless exponential steps are possible.*

▶ **Proposition 11** (Usefulness of exponential steps). *Let* $p = P\langle x\rangle \to_{\mathsf{ue}} P\langle v\rangle = q$ *with* $P \in \mathcal{E}^{@}_{\mathcal{U},\mathcal{A}}$ *and* $P(x) = v$*. Then there exists a program* r *and a reduction sequence* $d : q \to^{k}_{\mathsf{ue}} \to_{\mathsf{um}} r$ *s.t.:*
1. *the evaluation context of each* $\to_{\mathsf{ue}}$ *steps in* d *is in* $\mathcal{E}^{@}_{\mathcal{U},\mathcal{A}}$*, and the one of* $\to_{\mathsf{um}}$ *is in* $\mathcal{E}_{\mathcal{U},\mathcal{A}}$*.*
2. $k \geq 0$ *is the number of* $\mathsf{E_{NA}}$ *rules in the derivation of* $P \in \mathcal{E}^{@}_{\mathcal{U},\mathcal{A}}$*.*

Completeness amounts to proving that useful normal forms, when unshared, give a Open CbNeed normal term. The point is that useful substitutions, if erroneously designed, might stop too soon, on programs that still contain – up to unsharing – some redexes. Completeness is developed in the following paragraph about useful normal forms.

Useful Normal Forms. We are now going to develop an inductive description of useful normal forms, that is, programs that are $\to_{\mathsf{und}}$-normal. The key property guiding the characterization of a useful normal program p is that if the sharing in p is unfolded (by turning ES into meta-level substitutions and obtaining a term) it produces a normal term of the open system, where the unfolding operation is defined as follows:

$$\text{Unfolding of programs} \qquad (t,\epsilon)\!\downarrow \coloneqq t \qquad\qquad (t, e[x\leftarrow u])\!\downarrow \coloneqq (t,e)\!\downarrow\{x\leftarrow u\}$$

The characterization rests on 3 predicates, defined in Fig. 6, for programs unfolding to variables ($\mathsf{genVar}_x(p)$), values ($\mathsf{uabs}(p)$), and non-variable inert terms ($\mathsf{uinert}(p)$). Programs satisfying $\mathsf{genVar}_x(p)$ are called *generalized variable* of *(hereditary) head* variable x – we also write $\mathsf{genVar}_{\#}(p)$ to state that there exists $x \in \mathsf{Var}$ such that $\mathsf{genVar}_x(p)$. Programs satisfying $\mathsf{uabs}(p)$ (resp. $\mathsf{uinert}(p)$), instead, are *useful abstractions* (resp. *useful inerts*). The predicate $\mathsf{unorm}(p)$ holds for programs satisfying either of the three described predicates, which we shall show being exactly programs that are normal in Useful Open CbNeed.

Generalized variables play a special role, because they can be extended to unfold to values or non-variable inert terms, by appending an appropriate ES to their environment with rule $\mathsf{A_{GV}}$ or $\mathsf{I_{GV}}$. For instance, a useful normal program such as $(x, [x\leftarrow y])$ unfolds to a variable but its useful normal extension $(x, [x\leftarrow y][y\leftarrow \mathsf{I}])$ unfolds to the value I, while $(x, [x\leftarrow y][y\leftarrow zz])$ unfolds to the non-variable inert term zz.

▶ **Proposition 12** (Disjointness and unfolding of useful predicates). *For every program* p*,* at most one *of the following holds:* $\mathsf{genVar}_{\#}(p)$*,* $\mathsf{uabs}(p)$*, or* $\mathsf{uinert}(p)$*. Moreover,*
1. *If* $\mathsf{genVar}_x(p)$ *then* $p\!\downarrow = x$*.*
2. *If* $\mathsf{uabs}(p)$ *then* $p\!\downarrow$ *is a value.*
3. *If* $\mathsf{uinert}(p)$ *then* $p\!\downarrow$ *is a non-variable inert term.*

While the concepts in the characterization of useful normal programs are relatively simple and natural, the proof of the next proposition is long and tedious, because of the complex shape of useful evaluation contexts and of their parametrization, see the technical report [18].

▶ **Proposition 13** (Syntactic characterization of Useful Open CbNeed-normal forms). *Let* p *be a program. Then* p *is in* $\to_{\mathsf{und}}$*-normal form if and only if* $\mathsf{unorm}(p)$*.*

$$\frac{}{\mathsf{genVar}_x(x,\epsilon)}\ \mathsf{GV}_{\mathsf{AX}} \qquad \frac{\mathsf{genVar}_x(p)}{\mathsf{genVar}_y(p@[x{\leftarrow}y])}\ \mathsf{GV}_{\mathsf{HER}} \qquad \frac{\mathsf{genVar}_x(p) \quad z \neq x}{\mathsf{genVar}_x(p@[z{\leftarrow}t])}\ \mathsf{GV}_{\mathsf{GC}}$$

$$\frac{}{\mathsf{uabs}(v,\epsilon)}\ \mathsf{A}_{\mathsf{Lift}} \qquad \frac{\mathsf{genVar}_x(p)}{\mathsf{uabs}(p@[x{\leftarrow}v])}\ \mathsf{A}_{\mathsf{GV}} \qquad \frac{\mathsf{uabs}(p)}{\mathsf{uabs}(p@[x{\leftarrow}t])}\ \mathsf{A}_{\mathsf{GC}}$$

$$\frac{}{\mathsf{uinert}(i^+,\epsilon)}\ \mathsf{I}_{\mathsf{Lift}} \qquad \frac{\mathsf{genVar}_x(p)}{\mathsf{uinert}(p@[x{\leftarrow}i^+])}\ \mathsf{I}_{\mathsf{GV}} \qquad \frac{\mathsf{uinert}(p) \quad x \in \mathsf{nv}(p)}{\mathsf{uinert}(p@[x{\leftarrow}i])}\ \mathsf{I}_{\mathsf{I}}$$

$$\frac{\mathsf{uinert}(p) \quad x \in \mathsf{u}(p) \quad x \notin \mathsf{a}(p)}{\mathsf{uinert}(p@[x{\leftarrow}v])}\ \mathsf{I}_{\mathsf{U}} \qquad \frac{\mathsf{uinert}(p) \quad x \notin \mathsf{nv}(p)}{\mathsf{uinert}(p@[x{\leftarrow}t])}\ \mathsf{I}_{\mathsf{GC}}$$

$$\frac{\mathsf{uinert}(p) \vee \mathsf{uabs}(p) \vee \mathsf{genVar}_{\#}(p)}{\mathsf{unorm}(p)}\ \mathsf{unorm}_P$$

Figure 6 Predicates characterizing Useful Open CbNeed normal forms.

The characterization of useful normal forms together with the fact that they unfold to Open CbNeed normal terms (Lemma 12) express the *completeness* of useful sharing: our useful evaluation does compute – up to unfolding – representations of normal terms.

Complexity. A precise complexity analysis requires an abstract machine implementing the search for redexes specified by evaluation contexts. The machine – which we have developed – is left to a forthcoming paper, for lack of space. Crucially, it avoids tracing sets of applied and unapplied variables by simply using a boolean that indicates – when evaluation moves into the environment – whether the current evaluation position is hereditarily applied.

We provide a sketch of the complexity analysis. The k in point 2 of Proposition 11 allows us to bound any sequence of consecutive $\to_{\mathsf{ue}}$ steps with the length of the environment, which – via the same reasoning used for the CbN case by Accattoli and Dal Lago [13] – gives a quadratic bound to the whole number of $\to_{\mathsf{ue}}$ steps in terms of $\to_{\mathsf{um}}$ steps. A finer amortized analysis, following Accattoli and Sacerdoti Coen [10], gives a linear bound. The cost of duplications in exponential steps $\to_{\mathsf{ue}}$ is bound by the size of the initial program, because the calculus evidently has the *subterm property* (i.e. only subterms of the initial programs are duplicated): it duplicates values but it does not substitute nor evaluate into them, therefore the initial ones are preserved. Then, the cost of implementing a reduction sequence $d : p \to^k_{\mathsf{und}} q$, omitting the cost of searching for redexes (itself usually realized linearly in the size $|p|$ of p by abstract machines [6, 9]), is linear in $|p|$ and in the number $|d|_{\mathsf{m}}$ of multiplicative/β steps in d.

Therefore, the number of multiplicative/β steps in our Useful Open CbNeed calculus is a reasonable time cost model, even realizable within an efficient, bilinear overhead.

References

1 Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Inf. Comput.*, 105(2):159–267, 1993.

2 Beniamino Accattoli. The complexity of abstract machines. In *WPTE@FSCD 2016*, pages 1–15, 2016. doi:10.4204/EPTCS.235.1.

3 Beniamino Accattoli. The Useful MAM, a Reasonable Implementation of the Strong λ-Calculus. In *WoLLIC 2016*, pages 1–21, 2016. doi:10.1007/978-3-662-52921-8_1.

4 Beniamino Accattoli. Proof nets and the linear substitution calculus. In Bernd Fischer and Tarmo Uustalu, editors, *Theoretical Aspects of Computing – ICTAC 2018 – 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings*, volume 11187 of *Lecture Notes in Computer Science*, pages 37–61. Springer, 2018. doi:10.1007/978-3-030-02508-3_3.

5 Beniamino Accattoli. A fresh look at the lambda-calculus (invited talk). In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, pages 1:1–1:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.FSCD.2019.1.

6 Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. Distilling abstract machines. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 363–376. ACM, 2014. doi:10.1145/2628136.2628154.

7 Beniamino Accattoli and Bruno Barras. Environments and the complexity of abstract machines. In *PPDP 2017*, pages 4–16, 2017. doi:10.1145/3131851.3131855.

8 Beniamino Accattoli and Bruno Barras. The negligible and yet subtle cost of pattern matching. In Bor-Yuh Evan Chang, editor, *Programming Languages and Systems – 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*, volume 10695 of *Lecture Notes in Computer Science*, pages 426–447. Springer, 2017. doi:10.1007/978-3-319-71237-6_21.

9 Beniamino Accattoli and Claudio Sacerdoti Coen. On the relative usefulness of fireballs. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 141–155. IEEE Computer Society, 2015. doi:10.1109/LICS.2015.23.

10 Beniamino Accattoli and Claudio Sacerdoti Coen. On the value of variables. *Inf. Comput.*, 255:224–242, 2017. doi:10.1016/j.ic.2017.01.003.

11 Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. Strong call-by-value is reasonable, implosively. In *LICS*, pages 1–14. IEEE, 2021.

12 Beniamino Accattoli, Andrea Condoluci, Giulio Guerrieri, and Claudio Sacerdoti Coen. Crumbling abstract machines. In Ekaterina Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, pages 4:1–4:15. ACM, 2019. doi:10.1145/3354166.3354169.

13 Beniamino Accattoli and Ugo Dal Lago. On the invariance of the unitary cost model for head reduction. In *RTA*, volume 15 of *LIPIcs*, pages 22–37. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012.

14 Beniamino Accattoli and Ugo Dal Lago. (Leftmost-Outermost) Beta-Reduction is Invariant, Indeed. *Logical Methods in Computer Science*, 12(1), 2016. doi:10.2168/LMCS-12(1:4)2016.

15 Beniamino Accattoli and Giulio Guerrieri. Open call-by-value. In Atsushi Igarashi, editor, *Programming Languages and Systems – 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 206–226, 2016. doi:10.1007/978-3-319-47958-3_12.

16 Beniamino Accattoli and Giulio Guerrieri. Abstract machines for open call-by-value. *Sci. Comput. Program.*, 184, 2019. doi:10.1016/j.scico.2019.03.002.

17 Beniamino Accattoli, Giulio Guerrieri, and Maico Leberle. Types by need. In Luís Caires, editor, *Programming Languages and Systems – 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 410–439. Springer, 2019. doi:10.1007/978-3-030-17184-1_15.

18 Beniamino Accattoli and Maico Leberle. Useful open call-by-need. *CoRR*, abs/2107.06591, 2021. URL: https://arxiv.org/abs/2107.06591, arXiv:2107.06591.

19 Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1997. URL: `http://journals.cambridge.org/action/displayAbstract?aid=44089`.

20 Zena M. Ariola, Hugo Herbelin, and Alexis Saurin. Classical call-by-need and duality. In C.-H. Luke Ong, editor, *Typed Lambda Calculi and Applications – 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, volume 6690 of *Lecture Notes in Computer Science*, pages 27–44. Springer, 2011. `doi:10.1007/978-3-642-21691-6_6`.

21 Andrea Asperti and Harry G. Mairson. Parallel beta reduction is not elementary recursive. *Inf. Comput.*, 170(1):49–80, 2001. `doi:10.1006/inco.2001.2869`.

22 Thibaut Balabonski. Weak optimality, and the meaning of sharing. In *ICFP*, pages 263–274. ACM, 2013.

23 Thibaut Balabonski, Pablo Barenbaum, Eduardo Bonelli, and Delia Kesner. Foundations of strong call by need. *PACMPL*, 1(ICFP):20:1–20:29, 2017. `doi:10.1145/3110264`.

24 Thibaut Balabonski, Antoine Lanco, and Guillaume Melquiond. A strong call-by-need calculus. In *FSCD*, volume 195 of *LIPIcs*, pages 9:1–9:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

25 Pablo Barenbaum, Eduardo Bonelli, and Kareem Mohamed. Pattern matching and fixed points: Resource types and strong call-by-need: Extended abstract. In David Sabel and Peter Thiemann, editors, *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*, pages 6:1–6:12. ACM, 2018. `doi:10.1145/3236950.3236972`.

26 Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1984.

27 Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7, 1999.

28 Malgorzata Biernacka and Witold Charatonik. Deriving an abstract machine for strong call by need. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, pages 8:1–8:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.FSCD.2019.8`.

29 Tomasz Blanc, Jean-Jacques Lévy, and Luc Maranget. Sharing in the weak lambda-calculus. In *Processes, Terms and Cycles*, volume 3838 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2005.

30 Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In John Williams, editor, *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pages 226–237. ACM, 1995. `doi:10.1145/224164.224210`.

31 Stephen Chang and Matthias Felleisen. The call-by-need lambda calculus, revisited. In Helmut Seidl, editor, *Programming Languages and Systems – 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 – April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 128–147. Springer, 2012. `doi:10.1007/978-3-642-28869-2_7`.

32 Arthur Charguéraud and François Pottier. Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In *ITP 2015*, pages 137–153, 2015. `doi:10.1007/978-3-319-22102-1_9`.

33 Ugo Dal Lago and Simone Martini. Derivational complexity is an invariant cost model. In Marko C. J. D. van Eekelen and Olha Shkaravska, editors, *Foundational and Practical Aspects of Resource Analysis – First International Workshop, FOPARA 2009, Eindhoven, The Netherlands, November 6, 2009, Revised Selected Papers*, volume 6324 of *Lecture Notes in Computer Science*, pages 100–113. Springer, 2009. `doi:10.1007/978-3-642-15331-0_7`.

34 Ugo Dal Lago and Simone Martini. On constructor rewrite systems and the lambda calculus. *Logical Methods in Computer Science*, 8(3), 2012. doi:10.2168/LMCS-8(3:12)2012.

35 Olivier Danvy and Ian Zerny. A synthetic operational account of call-by-need evaluation. In Ricardo Peña and Tom Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 97–108. ACM, 2013. doi:10.1145/2505879.2505898.

36 Paul Downen, Luke Maurer, Zena M. Ariola, and Daniele Varacca. Continuations, processes, and sharing. In Olaf Chitil, Andy King, and Olivier Danvy, editors, *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, pages 69–80. ACM, 2014. doi:10.1145/2643135.2643155.

37 Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. *Log. Methods Comput. Sci.*, 6(3), 2010. URL: http://arxiv.org/abs/1003.5197.

38 Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, pages 235–246. ACM, 2002. doi:10.1145/581478.581501.

39 Jennifer Hackett and Graham Hutton. Call-by-need is clairvoyant call-by-value. *Proc. ACM Program. Lang.*, 3(ICFP):114:1–114:23, 2019. doi:10.1145/3341718.

40 Hugo Herbelin and Étienne Miquey. A calculus of expandable stores: Continuation-and-environment-passing style translations. In *LICS*, pages 564–577. ACM, 2020.

41 Delia Kesner. Reasoning about call-by-need by means of types. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures – 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9634 of *Lecture Notes in Computer Science*, pages 424–441. Springer, 2016. doi:10.1007/978-3-662-49630-5_25.

42 Delia Kesner, Loïc Peyrot, and Daniel Ventura. The spirit of node replication. In *FoSSaCS*, volume 12650 of *Lecture Notes in Computer Science*, pages 344–364. Springer, 2021.

43 Delia Kesner, Alejandro Ríos, and Andrés Viso. Call-by-need, neededness and all that. In *FoSSaCS*, volume 10803 of *Lecture Notes in Computer Science*, pages 241–257. Springer, 2018.

44 Arne Kutzner and Manfred Schmidt-Schauß. A non-deterministic call-by-need lambda calculus. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, pages 324–335. ACM, 1998. doi:10.1145/289423.289462.

45 John Launchbury. A natural semantics for lazy evaluation. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 144–154. ACM Press, 1993. doi:10.1145/158511.158618.

46 Maico Leberle. *Dissecting call-by-need by customizing multi type systems*. Theses, Institut Polytechnique de Paris, May 2021. URL: https://tel.archives-ouvertes.fr/tel-03284370.

47 Jean-Jacques Lévy. Réductions correctes et optimales dans le lambda-calcul. Thése d'Etat, Univ. Paris VII, France, 1978.

48 John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, 1998. URL: http://journals.cambridge.org/action/displayAbstract?aid=44169.

49 Masayuki Mizuno and Eijiro Sumii. Formal verifications of call-by-need and call-by-name evaluations with mutual recursion. In *APLAS*, volume 11893 of *Lecture Notes in Computer Science*, pages 181–201. Springer, 2019.

50 Pierre-Marie Pédrot and Alexis Saurin. Classical by-need. In Peter Thiemann, editor, *Programming Languages and Systems – 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 616–643. Springer, 2016. doi:10.1007/978-3-662-49498-1_24.

51 David Sands, Jörgen Gustavsson, and Andrew Moran. Lambda calculi and linear speedups. In Torben Æ. Mogensen, David A. Schmidt, and Ivan Hal Sudborough, editors, *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, volume 2566 of *Lecture Notes in Computer Science*, pages 60–84. Springer, 2002. doi:10.1007/3-540-36377-7_4.

52 Peter Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3):231–264, 1997. URL: http://journals.cambridge.org/action/displayAbstract?aid=44087.

53 Christopher P. Wadsworth. *Semantics and pragmatics of the lambda-calculus*. PhD Thesis, Oxford, 1971.