Sharing a Perspective on the lambda Calculus

Beniamino Accattoli

Sharing a Perspective on the λ-Calculus

Beniamino Accattoli
beniamino.accattoli@inria.fr
Inria & Lix, École Polytechnique
Palaiseau, France

Abstract

The λ-calculus models the core of functional programming languages. This essay discusses a gap between the theory of the λ-calculus and functional languages, namely the fact that the former does not give a status to *sharing*, the essential ingredient for efficiency in the lattter.

The essay provides an overview of the perspective of the author, who has been and still is studying sharing from various angles. In particular, it explains how sharing impacts the *equational* and *denotational* semantics of the λ-calculus, breaking some expected properties, and demanding the development of new, richer semantics of the λ-calculus.

CCS Concepts: • **Theory of computation → Lambda calculus**.

Keywords: Lambda calculus, functional languages, sharing

ACM Reference Format:
Beniamino Accattoli. 2023. Sharing a Perspective on the λ-Calculus. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '23), October 25–27, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3622758.3622884

Foreword. This essay addresses students and researchers with a background in functional programming languages or in the theory of λ-calculus. The aim is to communicate a few important ideas about sharing in these areas while avoiding diving into complex technical depth. Another aim is to reach both theoreticians interested in denotational models and implementors interested in abstract machines, who often belong to separate communities and use different terminologies. Lastly, the essay is not meant to be an exhaustive overview of sharing, neither at the level of concepts or bibliographic references. It instead focusses on the insights emerging from the work developed by the author in the last decade.

Onward! '23, October 25–27, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0388-1/23/10…$15.00
https://doi.org/10.1145/3622758.3622884

1 Introduction

Functional languages are a family of powerful programming languages enhancing modularity and reducing bugs, while being based on solid mathematical grounds. Besides being used to develop software, these languages play a crucial role in the theory of proof assistants such as Coq, Agda, or Isabelle, which are increasingly applied to the *verification* of software and mathematics.

The core of functional languages is mathematically modeled by the λ-calculus. The starting point of this essay is the observation that there is too much of a gap between the theory of λ-calculus and the study of functional languages.

Historically, such a remark is not novel. In 1993, Abramsky and Ong pointed out an *evaluation gap* [3], stressing that functional languages are based on *weak evaluation* within the λ-calculus, that is, they do not evaluate function bodies (more precisely, they evaluate them only if the function receives its arguments), while the theory of the λ-calculus used to be based on *strong evaluation*, which does allow one to evaluate function bodies (that is, strong evaluation also evaluates the body of functions that are defined but never applied to arguments). Their work has been hugely influential, and 30 years later the theoretical study of the weak evaluation has considerably developed. Additionally, strong evaluation is now applied more often, namely in the theory of proof assistants based on dependent types, such as Coq or Agda.

The gap we refer to, however, is of a different nature. It does concern evaluation, but not *what* is evaluated, rather *how* it is evaluated, thus it concerns both the weak and the strong settings. The λ-calculus does not capture two essential aspects of functional languages, namely the *efficiency* and the many *programming features* of an actual language. The features gap is somewhat intrinsic to the idea of reducing a rich programming language to a barebone mathematical setting such as the λ-calculus, and we shall pay no attention to it here. The aim of this essay is to explain how the efficiency gap, often considered a theoretically negligible aspect, leads to a considerably richer theory when taken seriously, and a theory that is far from being solid and fully developed.

An Analogy. The problem can actually be explained via an evocative analogy. Let us think of a programming language as of a complex piece of craftsmanship such as a luxury car, which has to be both powerful and comfortable for the

programmer. The engine of a programming language is how functions, procedures, and methods are defined, how they can be called and composed, and how such mechanisms are implemented. This engine is what is modeled by the λ-calculus. Except that, for the sake of theoretical clarity, the λ-calculus is a basic form of engine, like a *steam engine*. In the programming landscape, languages such as OCaml or Haskell can be compared to Tesla cars that are nowadays often on the news: a minoritarian approach, spreading quickly, and based on next-generation technology (the functional aspect). For Theory to catch up to Practice, and possibly improve it, two main tasks need to be addressed: upgrading the theoretical engine of reference from *steam* to electric (the efficiency gap), and moving from having an efficient engine and producing a luxury car (the features gap).

Sharing. Let us translate the analogy into more accurate scientific terms. Most theoretical studies in λ-calculus still focus on its *call-by-name* version introduced almost a century ago by Church. This framework is unsuitable to modern applications, as it does not model sharing, the mandatory ingredient for efficiency. In practice, one needs to consider Plotkin's *call-by-value* λ-calculus [54] or Wadsworth's *call-by-need* λ-calculus [56], that do model sharing—these shall here be called *λ-calculi with sharing* and constitute the electric engine mentioned above. Unfortunately, their foundations are much less developed, because they are richer and more challenging settings, and cannot be treated as variants of the call-by-name λ-calculus. Notably, adopting the same technical notion of meaningful program used by Barendregt in his famous book on the (call-by-name) λ-calculus [25], leads to an *inconsistent theory* in call-by-value, see Accattoli and Guerrieri [18].

2 A Bit of Context

In the theory of programming languages, the functional—or *higher-order*—approach studies a notion of program for which the inputs and the outputs are not simply numbers, strings, or compound data types, but may be *programs themselves*. Moreover, this paradigm sees programs as functions and provides flexible ways to define them, in particular allowing one to define *nested* functions (that is, functions defined inside the definition of other functions, which can refer to the variables of the enclosing functions), which is usually not possible in non-functional languages.

Such an approach is as old as computer science, being based on Church's *λ-calculus*, which is one of the early attempts—together with Turing's machines, and Gödel's partial recursive functions—at formalizing computations, based solely on the concept of (nested) functions.

For a long time, higher-order features were only considered in programming languages coming from the academic world, such as LISP, λ-prolog, Haskell, or OCaml. The situation has however changed. Firstly, the use of Haskell and OCaml has spread out of academia. For instance, OCaml is used in the web version of Facebook Messenger (see this link[1]) or by financial companies such as Jane Street Capital for developing their high-frequency trading software. Secondly, mainstream languages have also been extended with higher-order features: starting from Java 8 (2014), indeed, Java has *λ-expressions*, that support higher-order computations, and languages such as Python or Scala made higher-order a key ingredient of their design.

Theory* vs *Practice. The study of the λ-calculus is often too detached from the real world. Most theoretical studies address the *call-by-name* (shortened to CbN) λ-calculus because it is the simplest framework to test new approaches and it has well-established links with other areas of mathematics like universal algebra or topology. However, it is *never* used in real-life applications, because of its intrinsic *inefficiency*.

The frameworks used for modeling functional languages and proof assistants are rather the *call-by-value* (shortened to CbV) and *call-by-need* (CbNeed) λ-calculi. For instance, *OCaml* is based on CbV, *Haskell* on CbNeed, and *Coq* uses both mechanisms. These calculi model more efficient higher-order computations but are based on much less developed theories. Even if these calculi are at the cores of practical tools, further extensions with pattern matching, inductive datatypes, recursion, effects, etc., must be considered to close the gap with such tools. Therefore, rather than being the skeleton of practical tools, they rather provide a refined engine still far from a real language.

Semantics of λ-Calculi. Higher-order languages originated as mathematical objects and they have often gone hand in hand with the development of their mathematical foundations. Such a formal study can be done from different points of view. To properly explain the current understanding of λ-calculi with sharing, let us briefly overview these points of view.

1. *Operational*: the operational viewpoint defines the program evaluation process and studies its *rewriting properties* such as confluence and normalization. It also concerns the study of evaluation strategies and of their properties such as standardization or factorization. One might also include in this approach the study of decompositions of the evaluation process such as *abstract machines*, and refinements of the study of termination into cost and *complexity analyses*.
2. *Denotational*: a denotational model is given by a family of mathematical objects—such as relations, domains, or games—into which λ-terms are mapped, in such a way that the evaluation process on terms corresponds

[1] https://reasonml.github.io/blog/2017/09/08/messenger-50-reason.html

to *equality on objects*, which are then invariants of evaluation. The requirements for such models are usually formulated in the language of *category theory*.

3. *Equational*: this form of semantics is the study of the many possible notions of *program equivalence* between terms—also called *equational theories*—such as contextual equivalence, notions of bisimilarities, or the identifications induced by denotational models. The properties of interest are *consistency* and the *entailments* between different equivalences, as well as the ease in establishing the equivalence in concrete cases. Equational semantics can be studied operationally (usually via notions of bisimulations) or denotationally (by characterizing the equational theory of denotational models). Equational semantics thus falls somewhere *between* the two previous semantics, and it is often entangled with them.
4. *Logical*: according to the *Curry-Howard correspondence* between λ-calculi and proof systems, terms of λ-calculi can be seen as proofs in fragments, extensions, or refinements of *intuitionistic logic*, with logical formulas playing the role of *types* for programs. The logical semantics is finer than the denotational one because there is a logical process mimicking program evaluation, called *cut-elimination*, rather than collapsing it on equality as in models.

3 Introducing First-Class Sharing

In the ordinary λ-calculus, a program can duplicate or erase a whole sub-program in a single macro computational step, called β-*reduction*. Let us set some basic notions:

$$\begin{array}{rrcl} \lambda\text{-Terms} & t,s & ::= & x \mid \lambda x.t \mid ts \\ \beta\text{-reduction} & (\lambda x.t)s & \to_\beta & t\{x\leftarrow s\} \end{array}$$

Abstractions $\lambda x.t$ bind x in t, and terms are identified up to α-renaming of bound variables. The notation $t\{x\leftarrow s\}$ indicates the (meta-level and capture-avoiding) substitution of s for x in t. Duplication and erasure of a sub-program by β-reduction are for instance given by the following examples:

Duplication	Erasure
$(\lambda x.xx)t \to_\beta tt$	$(\lambda x.y)t \to_\beta y$

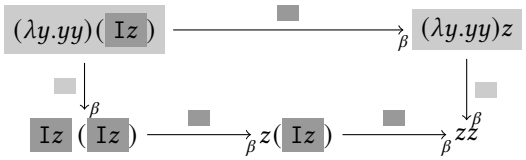
In the theory of the λ-calculus developed until the 1990s—typically the one in Barendregt's book [25], which is the classical reference about the λ-calculus—the β-reduction rule can be applied anywhere in a term, that is, whenever a term u has a sub-term of shape $(\lambda x.t)s$, which is called a β-*redex*, then the sub-term can be rewritten according to the β-rule. This is nowadays referred to as the *strong* λ-*calculus*. In the already mentioned *weak* λ-*calculus*, instead, β-reduction cannot take place in abstractions. In the strong λ-calculus, a *normal form* is a term without β redexes, while in the weak λ-calculus it is a term having no β-redexes outside of abstractions. For instance, $\lambda z.((\lambda x.t)s)$ is a normal form in the weak setting but not in the strong one, where it β-reduces to $\lambda z.(t\{x\leftarrow s\})$.

Values and Sharing. The CbV and CbNeed variants avoid the inefficient repetition of work—typical of CbN—by limiting what can be duplicated, namely, only *values*, a restricted form of sub-program.

$$\begin{array}{rrcl} \text{Values} & v & ::= & x \mid \lambda x.t \\ \text{CbV } \beta\text{-reduction} & (\lambda x.t)v & \to_{\beta_v} & t\{x\leftarrow v\} \end{array}$$

The CbV restriction forces sub-programs to be duplicated and erased only *after* they are evaluated. This restriction results in sharing of computations. The definition of CbNeed is omitted here, as it is a bit more technical, but we shall explain the intuition behind it in Sect. 7. Roughly, CbNeed also only duplicates values, but can instead *erase any term*.

Let us give an example of how the value restriction realizes sharing. Let $\mathtt{I} := \lambda x.x$ be the identity λ-term and consider the following two evaluation paths of $(\lambda y.yy)(\mathtt{I}z)$, where the horizontal arrows reduce the redexes highlighted in dark gray (which are all occurrences of $\mathtt{I}z$) while the vertical arrows reduce the redexes highlighted in light gray:



(1)

The top-right path (reducing the dark gray redex first) is a CbV evaluation. A sharing of computations is obtained via the value restriction which forces to reduce the (dark grey) redex before duplicating it, because $\mathtt{I}z$ is not a value. The left-bottom path is a CbN evaluation, which is longer than the CbV one, because it reduces each of the two copies of the redex $\mathtt{I}z$ produced by the light gray redex. If one assumes that efficiency can be measured by comparing the number of β-steps[2], then CbV is more efficient than CbN (in this case).

A Confusing Point. A strongly misleading point is that Plotkin's presentation of the CbV λ-calculus does not make sharing explicit via a term constructor (while it is usually explicit in the presentations of CbNeed due to Launchbury [41], Ariola and Felleisen [23], Maraist et al. [48], and Sestoft [55]). Under the assumptions used to model functional programming—namely, *closed terms* (that is, without free variables) and *weak evaluation*—sharing can indeed be swept under the rug. It turns out, however, that both semantical studies and proof assistants need to go beyond those assumptions by considering *open terms* (that is, terms with free variables; for instance $x(\lambda y.yz)$ is open because x and z are free). Then, the theory of CbV for the simplified case is *no longer adequate* and some representation of sharing becomes unavoidable, as discussed at length by Accattoli and Guerrieri [16].

[2]For CbV and CbN one can indeed compare the number of β-steps, but this is not true for general β-reduction sequences. Behind it lies the theory of reasonable time for the λ-calculus, see [7] for an introduction.

CbN Sharing* vs *CbV and CbNeed Sharing. *Sharing* is an overloaded word for many different aspects of the efficient evaluation of λ-terms. A form of sharing—namely sub-term sharing, discussed below—is also used in abstract machines for the CbN λ-calculus. There are two aspects, however, that set the by-value and by-need settings apart. Firstly, they also share *computations*, not just sub-terms (as diagram (1) above hints at, and as we shall better explain below). Secondly, they give a first-class status to sharing: not only it is used in the abstract machines, it is also part of the syntax (in Accattoli and Guerrieri's setting for CbV [16, 18]) and it does not vanish at the end of evaluation (as instead happens in CbN), *ending up in the results*. Therefore, we refer to the by-value and by-need settings together as *λ-calculi with sharing*.

Adding Sharing. Adopting a first-class treatment of sharing is delicate because it does not amount to simply *adding* something, as it would be the case with, say, pairs or continuations. It does indeed require a modification of the evaluation mechanism itself, and it is made up of four basic steps.

1. *Syntax*: first of all, the syntax is extended with a constructor such as let $x = s$ in t. In fact, let expressions are often more compactly represented as *explicit substitutions* (shortened to *ESs*) $t[x \leftarrow s]$, also to abandon the widespread convention that s evaluates before t in let $x = s$ in t, which depends on the choice of an evaluation strategy rather than on the representation of sharing.

   Terms with sharing
   $$t, s \quad ::= \quad x \mid \lambda x.t \mid ts \mid t[x \leftarrow s]$$
2. *Decomposing β*: next, one modifies β-reduction as to turn $(\lambda x.t)s$ into $t[x \leftarrow s]$ which is an annotation *delaying* the substitution $t\{x \leftarrow s\}$, thus keeping s *shared* instead of duplicating it.
3. *Rewriting ESs*: the next natural step is to introduce rewriting rules for $t[x \leftarrow s]$. There are many possible such sets of rules. In the simplest case, one has a small-step CbN rule:
   $$t[x \leftarrow s] \to_{\mathsf{sub}} t\{x \leftarrow s\}$$
   Usually, however, one has a set of *micro-step* rules making *one copy of s at a time*, rather than all at once, and—in CbV and CbNeed—only values can be substituted (that is, s has to be a value in rule $\to_{\mathsf{sub}}$ above, and in its micro-step variants). In this essay, we prefer to avoid spelling out a set of rewriting rules for ESs, in order to keep the discussion high-level and informal, at the price of being vague, but also because sharing and its operational semantics admit various presentations (many using terms, but also via graphs, abstract machines, etc), and we do not want the reader to be misled by the details of a fixed presentation.
4. *Evaluation contexts*: finally, one needs to establish in which *evaluation contexts* the rules can be applied. The choice of rules and of evaluation contexts defines the *evaluation strategy* of the chosen λ-calculus with sharing. A key aspect determined by the evaluation contexts is whether evaluation goes under abstraction: if it does not, then the calculus is *weak*, as it is usually the case in functional languages, otherwise it is *strong*, which is mainly used in the theory of proof assistants.

Depending on the rules for ES and on the choice of evaluation contexts, there is in fact a whole *family* of λ-calculi with sharing, depending at least on whether the calculus is weak or strong, the granularity of the rewriting rules is small-step or micro-step, and whether evaluation is by-name, by-value, or by-need.

Three Levels of Sharing. With the introduced concepts, we can already identify three levels or aspects of sharing (further forms of sharing exist, but we are not aiming at a comprehensive survey on sharing here):

- *Sharing of sub-terms*: the ES constructor $t[x \leftarrow s]$ is an explicit annotation for the sharing of the sub-term s in t. If evaluation does *not* touch s, but only keeps it shared (until it is substituted, if ever), then there is *no sharing of computations*. Such a sub-term sharing is what is found in the abstract machines for CbN, and it is already a powerful tool, as it allows one, in some cases, to achieve an *exponential compression* of terms. For instance, consider the term:
  $$t_3 := (\lambda x.y_1y_1)[y_1 \leftarrow y_2y_2][y_2 \leftarrow y_3y_3]$$
  If we remove the sharing from t_3 by turning ESs into meta-level substitutions—an operation noted $\cdot\downarrow$ and usually called *(sharing) unfolding*—we obtain the term:
  $$t_3\downarrow \; = (\lambda x.y_3y_3y_3y_3y_3y_3y_3y_3)$$
  Clearly, generalizing such a process to $n > 3$ shows that $t_n\downarrow$ has size *exponentially bigger* than t_n. Moreover, in a weak setting (where evaluation does not go under abstraction, and so the substitutions on y_1 are never performed) both $t_n\downarrow$ and t_n are normal terms, so that t_n is a very compact shared representation of the sharing-free normal form $t_n\downarrow$.
- *Sharing of computations*: sharing is pushed one level further if one allows evaluation to happen inside s in $t[x \leftarrow s]$. Such an approach induces *sharing of evaluation sequences*, that is, of *computations*, which can then become exponentially shorter in some cases. For instance, consider (where $\mathtt{I}$ is the identity λ-term):
  $$s := (y_1y_1)[y_1 \leftarrow y_2y_2] \dots [y_{n-1} \leftarrow y_ny_n][y_n \leftarrow \mathtt{I}(zz)]$$
  In s, if reduction can enter $[y_n \leftarrow \mathtt{I}(zz)]$ then the redex $\mathtt{I}(zz)$ is reduced only once. Otherwise (typically in CbN), the evaluation of s might do up to 2^n copies of $\mathtt{I}(zz)$ (depending on whether the CbN evaluation strategy evaluates arguments), which then requires reducing that same redex an exponential number of times.

- *Sharing of computations by value/need*: in CbV and CbNeed settings, there is sharing of computations *plus* the constraint that only values can be duplicated, that is, an ES not containing a value shall persist. The constraint has an important consequence: on some terms, sharing cannot be eliminated (even when evaluating inside abstractions). For instance, in CbV or CbNeed the term s above would reduce to the following one:
$$(y_1 y_1)[y_1 \leftarrow y_2 y_2] \dots [y_{n-1} \leftarrow y_n y_n][y_n \leftarrow zz]$$
and in this term none of the ESs contains a value, so that the term is normal, and has ESs in its normal form.

Normal Forms, Unfolding, and Relational Models. The presence of sharing in normal forms in an important aspect that might be easily misunderstood. Depending on the definition of the operational semantics, also CbN might have ESs in normal forms. The difference is that in CbN it is *always sound* to unfold ESs and obtain an ordinary λ-term without ES. In CbV and CbNeed, instead, unfolding the sharing *can change the semantics*, as we now point out.

An important family of models of the λ-calculus is the *relational semantics*, which exists for CbN (this model is folklore, it is used for instance in de Carvalho [31]), CbV (introduced by Ehrhard [33]), and CbNeed (by Accattoli et al. [19]), and it is a paradigmatic family of models in that other families are built on top of it (such as coherence spaces, hypercoherences, finiteness spaces) or are strongly related to it (such as game models). Relational semantics easily model ESs. In the CbN relational semantics, t and its sharing unfolding $t\downarrow$ have the *same interpretation*, for every term t. In the CbV and CbNeed relational semantics, instead, t and $t\downarrow$ have in general *different interpretations*.

Therefore, in CbV and CbNeed sharing has a (denotational and equational) semantical role that is instead *invisible* in CbN.

4 Sharing, Operationally

The operational study of λ-calculi with sharing has been an active research topic for decades, because it was far from evident what was the right approach to the third step to the recipe for adding sharing given in the previous section (namely, "rewriting ES"). Among the various reasons, one might mention the *hard-to-manage graphical formalisms* of Wadsworth's original presentation of CbNeed [56], or the fact that natural choices of micro-step rules can introduce *malicious behavior*, breaking key properties of the λ-calculus, as shown by Melliès [49].

Nowadays, however, the operational semantics of sharing rests on *solid grounds*. The long-standing problem of finding canonical micro-step rules for ESs, indeed, has arguably been solved, namely via the *linear substitution calculus* [4, 10] (shortened to LSC) due to Accattoli and Kesner, merging ideas by Robin Milner [50] with insights from graphical languages and linear logic from their previous work [20]. The next paragraph gives some pointers to the literature on the LSC, for the interested reader.

The LSC. The LSC is today considered the standard of reference for λ-calculi with sharing because of its *unique rewriting properties* [10], its connections with *abstract machines* [9], and the theory of reasonable time that it enabled. In 2014, indeed, Accattoli and Dal Lago showed that the number of β-steps (of the leftmost strategy) is a reasonable time cost model for the strong λ-calculus [13]—the first such one, solving a long-standing open problem—via a fine study of sharing in the LSC.

A time or space cost model is *reasonable* if it is equivalent to the one of Turing machines, thus preserving the corresponding notion of computational complexity. Before 2014, only partial results about reasonable time for the λ-calculus were known. Since [13], Accattoli and co-authors have developed a solid theory of reasonable cost models. In 2022, Accattoli, Dal Lago, and Vanoni provided also the first reasonable space cost model accounting for *logarithmic space* [14]—solving the other long-standing open problem of the area—via a fine study of sharing in abstract machines.

The CbV variant of the LSC, the *value substitution calculus* (VSC), is the main tool in Accattoli and Guerrieri's effort in refining Plotkin's CbV as to overcome its defects with respect to open terms and provide a CbV theory of meaningful terms [16, 18, 22, 27]. They also used it to study abstract machines and reasonable time for CbV [11, 12, 17].

The LSC has also been used by Accattoli, Kesner, and coauthors to revisit, simplify, and extend *the theory of CbNeed* [9, 21, 24, 38, 39] and generalized as to encompass *linear logic* [8].

5 Sharing, Denotationally and Equationally

The denotational and equational semantics of sharing, however, are far *less understood*. One of the reasons, is that the semantics of sharing has mainly been studied in CbN, where sharing does not appear on normal forms. Another already mentioned one is the fact that sharing does not have a first-class status in Plotkin's presentation of CbV. To better explain the equational issue, we need to recall a key concept.

Contextual Equivalence. The paradigmatic notion of *equational semantics* for λ-calculi is Morris' *contextual equivalence* $\simeq_C$ [51], stating that two λ-terms t and s are equivalent if they both terminate or both diverge (with respect to a fixed notion of termination) when *plugged in the same context*, that is, that $C\langle t\rangle \Downarrow$ if and only if $C\langle s\rangle \Downarrow$ for all contexts C. It is the challenging benchmark, or the golden standard, for denotational models: a model is *fully abstract* if all contextually equivalent terms, and only them, have the same interpretation in the model.

Contextual equivalence is, however, a notion that is very difficult to manipulate, because of the quantification over all

contexts. This is why finding alternative characterizations (via models or bisimilarities), or sub-equivalences that are easier to manipulate, is of great importance.

Meaningful Programs. One of the first insights of the equational and denotational semantics of the λ-calculus has been the distinction between *idle looping programs* and *programs that diverge while having a productive behavior*, e.g., programs producing streams of data. The former are considered *meaningless*, have a degenerated representation in models, and are *all contextual equivalent*. The latter are instead considered *meaningful* and are represented as non-trivial elements of models, on par with terminating programs. Moreover, meaningful programs are *not* contextually equivalent unless they have the same productive behavior, and an equational theory identifying two of them is usually *inconsistent* (that is, it equates all terms). Barendregt [25] has built the theory of CbN λ-calculus, where meaningful/meaningless corresponds to the technical notion of *solvable/unsolvable*, out of this insight.

In CbV, the situation is different: unsolvable terms are *not all contextually equivalent* and equating them as in CbN leads to *inconsistency* [18]. Inspired by the LSC, a decade ago Accattoli and Guerrieri explored variants of Plotkin's CbV to circumvent some of its semantical shortcomings [22, 27]. Thanks to these insights, they recently found a *finer* notion of meaningful CbV program, namely scrutable terms [18], the dual notion of which—that is, *inscrutable terms*—are all contextually equivalent. This fact shows that the equational semantics of CbV is finer than the CbN one, despite being much less developed. It clearly calls for *more research* on the equational and denotational semantics of the CbV λ-calculus.

6 Trees *vs* DAGs, and Switchable Equivalences

In CbN, programs are syntax trees (with binders). Accordingly, many results in denotational semantics are based on a notion of "semantical trees" called *Böhm trees* [25], or some of their variants such as Lévy-Longo [44] or Nakajima trees [52]. Roughly, the idea is that semantical trees are potentially infinite extensions of normal forms taking into account the idle/productive behavior of diverging terms:

- *Terminating terms*: their semantical tree is their normal form;
- *Productive diverging terms*: their semantical tree is the infinite tree that they produce by pushing reduction to the limit;
- *Idle diverging terms*: their semantical tree is the special one-node $\bot$ semantical tree, which is a degenerate semantical tree representing lack of meaning.

Different notions of semantical tree (Böhm, Lévy-Longo, Nakajima) depend on the notion of termination taken as reference in the definition of semantical trees (those induced by head reduction, weak head reduction, head reduction plus η-equivalence, respectively).

From Trees to DAGs. CbV and CbNeed programs can also be seen as trees, and one can also define notions of CbV and CbNeed semantical trees. But sub-term sharing is better represented by *sharing* sub-trees, thus turning programs into directed acyclic graphs (DAGs), as we shall do below. The switch to DAGs can also be modelled without introducing a graphical formalism, by considering terms with sharing modulo a set of equivalences. The paradigmatic equivalence is the following *commutation of independent ESs*, where independence is given by the side conditions $x \notin \mathtt{fv}(u)$ and $y \notin \mathtt{fv}(s)$:

$$t[x{\leftarrow}s][y{\leftarrow}u] \quad \equiv_{\mathrm{com}} \quad t[y{\leftarrow}u][x{\leftarrow}s] \quad \begin{array}{l}\text{if } x \notin \mathtt{fv}(u)\\ \text{and } y \notin \mathtt{fv}(s).\end{array}$$

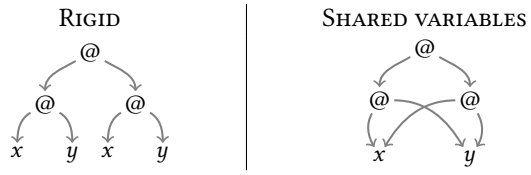
As an example, consider (where w is a variable):

$$\begin{array}{c}((xy)(xy))[x{\leftarrow}zz][y{\leftarrow}ww]\\ \equiv_{\mathrm{com}}\\ ((xy)(xy))[y{\leftarrow}ww][x{\leftarrow}zz].\end{array} \tag{2}$$

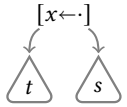
The idea of the commutation $\equiv_{\mathrm{com}}$ is that the position of an ES in a term does not matter, only the variable bound by the ES matters.

We can now use $\equiv_{\mathrm{com}}$ to better explain the tree and DAG representations of sharing via an informal discussion about graphical representations of terms.

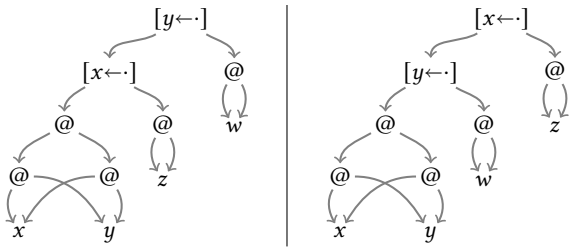
In a rigid tree representation, occurrences of a same variable are represented with different nodes. A first step in the representation of sharing is to keep the tree structure for the internal nodes, while using a single node for all the occurrences of the same variable. For instance, the two representations of $(xy)(xy)$ are:



We adopt the shared variables representation. Now, one might be tempted to represent the ES $t[x{\leftarrow}s]$ as a tree constructor as follows:

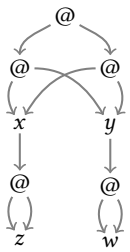


Such a representation gives different tree representations for the two terms in (2), as the root node of the first one would be labelled with $[y{\leftarrow}\cdot]$, while the root node of the second term would be $[x{\leftarrow}\cdot]$:



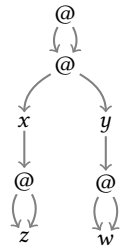
This is somewhat problematic, because in CbV and CbNeed, these two terms are both *normal*, since their ESs contain terms that are not values, and they have the *same interpretation* in the relational model. Thus one might want them to be represented by the *same* semantical tree (which, being normal terms, is just the tree representation of their normal form).

A better way to represent ESs, as already said, is exploiting DAGs. The idea is to change the representation of $t[x\leftarrow s]$ as to plug the representation of s under the node representing x, and *not* by adding a root node for the ES. With respect to the terms in (2), the modified translations maps them both to the following DAG:

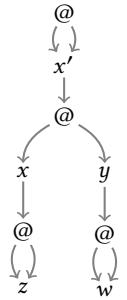


Therefore, the DAGs representation achieves the identification of the two semantic forms.

It is natural to want to push the DAGs representation further and also share application nodes, obtaining the following DAG:

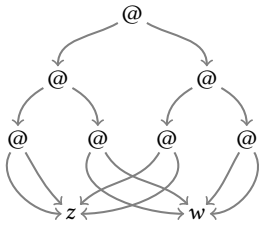


Such a representation is perfectly fine from the graphical point of view. It is however preferable to always associate sharing to variable nodes, as to preserve a tight connection between graphs and terms. In terms, indeed, sharing is encapsulated in the ES construct, which is always associated to a variable. Such a *sharing-on-variable-nodes* approach, represents the previous DAG as follows:



The corresponding term is $(x'x')[x'\leftarrow xy][x\leftarrow zz][y\leftarrow ww]$, which is neither of the terms in (2).

To connect with a previous section, let us discuss another way of associating a tree to the terms in (2). The idea is to unfold the sharing, which causes both terms to become $((zz)(ww))((zz)(ww))$. The graphical representation then becomes:



We already mentioned that unfolding is sound for CbN but not for CbV or CbNeed: in other words, this tree and the previous DAG have the same CbN semantics (in the CbN relational model) but not the same CbV or CbNeed semantics (in the respective relational models). Therefore, it would not be sound to use this tree as the CbV or CbNeed semantical tree of the terms in (2).

Beyond such an informal discussion, DAGs formalisms for λ-terms with sharing based on linear logic proof nets can be found in Accattoli [5, 6].

Switchable Equivalences. Let us now go back to the equivalence $\equiv_{\mathrm{com}}$ on terms. Since $\equiv_{\mathrm{com}}$ is symmetric, it *cannot be oriented* as a rewriting rule, and thus its equivalence classes do *not* admit *canonical representatives*. To obtain them, one needs the DAG representation sketched above. Alternatively, one simply forgets about canonical representatives and works modulo $\equiv_{\mathrm{com}}$ in the LSC, which has been conceived exactly for such a task. With DAGs or modulo $\equiv_{\mathrm{com}}$, notions of program equivalence become subtler, and the techniques developed for the tree-based setting of the CbN λ-calculus are often not adequate.

The equivalence $\equiv_{\mathrm{com}}$ is particularly interesting because it is valid in CbV without *effects*, and in particular it underlies the *parallel aspect* of CbV (such as the one used by Blelloch and Greiner [26] to connect the CbV λ-calculus with parallel random access machines), but it is *invalid* in CbV with *non-commutative effects* (such as forms of state), where a left-to-right or right-to-left evaluation order has to be adopted. On the other hand, it is valid in CbNeed (with or without

effects), since ESs in CbNeed are evaluated on-demand, independently of their syntactic order. Therefore, ideally, one might want models and program equivalences for which the validity of $\equiv_{\mathrm{com}}$ can be parametrically *switched on and off*, without having two whole different theories.

In CbN, the main degree of freedom in the design of models and program equivalences is the amount of *extensionality* (technically, η-equivalence) of the model/equivalence. In particular, there is an *axiomatisation of CbN contextual equivalence*, in the sense that it is known that CbN contextual equivalence amounts to $\beta\eta$-equivalence (more precisely η-equivalence of semantical trees).

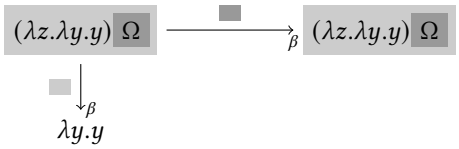
In CbV and CbNeed, the situation is different. In addition to β and η, there are equivalences such as $\equiv_{\mathrm{com}}$ and others that are discussed in the preprint by Accattoli et al. [15]. Therefore, the semantical account of λ-calculi with sharing has to be flexible, ideally producing *conceptual blocks* and techniques that can be *modularly composed* in order to model the λ-calculus with sharing of choice. Moreover, there is no known axiomatisation of CbV contextual equivalence, nor an accepted notion of semantical tree or DAG (there are some recent proposals [15, 46], but they are not fully satisfying).

Unifying Settings. In spite of the mentioned differences between CbN and CbV, there is a literature about dealing *uniformly* with some aspects of CbN and CbV λ-calculi. Such a literature is rooted in the classical-logic-based view of these paradigms, seeing them as opposite solutions to the failure of confluence for cut-elimination in *classical logic*. The uniform treatment is thus obtained via classically-inspired systems such as Levy's *call-by-push-value* [43] or Curien and Herbelin's $\bar{\lambda}\mu\tilde{\mu}$*-calculus* [29], or via linearly-inspired systems—because linear logic can represent classical logic—such as Laurent's *polarized logic* [42] or Ehrhard and Guerrieri's *bang calculus* [34]. These frameworks allow one to focus on the *similarities* rather than the differences between CbN and CbV. They uniformly capture the different evaluation orders in CbN and CbV, that is, their different selection of redexes. At present, they fail at capturing the *equational difference* between CbN and CbV, and they let *CbNeed* out of their unification. For that, another, different form of duality has to be involved, as we now explain.

7 Wise and Silly Duplication and Erasure

A crucial aspect in the construction of a theory of sharing is the management of *duplication* and *erasure*. In this respect, CbN and CbV are dual. This is *not*, however, the classical-logic-related duality between CbN and CbV, as this second duality can be observed also at the *intuitionistic level*, where CbV and CbNeed were originally introduced in the 1970s, more than a decade before the discovery of the computational context of classical logic by Griffin [36]. Therefore, it is somewhat *more primitive*.

CbN *never* evaluates arguments of β-redexes before the redexes themselves. As a consequence, it never evaluates in sub-terms that will be erased. This is *wise* with respect to erasure, and makes CbN a normalizing strategy, that is, a strategy that reaches a result whenever one exists. For instance, consider the following diagram, where Ω is the standard idling looping λ-term (defined as $\Omega := (\lambda x.xx)(\lambda x.xx)$ and verifying $\Omega \to_\beta \Omega$):



(3)

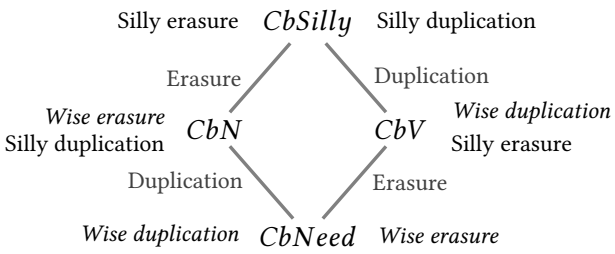
The vertical step (in light gray) is CbN, and erases the divergent argument without evaluating it.

A second consequence is that if the argument of the redex is duplicated then it may be evaluated more than once, as diagram (1) (page 3) shows. This is *silly* with respect to duplication, as it repeats work already done.

CbV, on the other hand, *always* evaluates arguments of β-redexes before the redexes themselves. Consequently, arguments are not re-evaluated. This is *wise* with respect to duplication, as diagram (1) shows, and it is essentially what accounts for sharing. But it has a consequence: arguments are also evaluated when they are going to be erased. For instance, in diagram (3) above, CbV evaluation performs the horizontal step (in dark gray) and *diverges* (as it keeps evaluating Ω) while CbN *terminates* in one β-step (simply erasing Ω). This CbV treatment of erasure is clearly as *silly* as the duplicated work of CbN.

CbNeed and its Silly Dual. It is natural to try to combine the advantages of both CbN and CbV. The strategy that is *wise* with respect to *both* duplications and erasures is nothing else but CbNeed: it does not evaluate arguments until they are needed (*wise erasure*), and, if they are needed for a second or further time, it re-uses the value previously calculated, avoiding the silly duplication of work of CbN (*wise duplication*). Such a view of CbNeed is used by Accattoli et al. [19] to develop a multi type system for CbNeed able to capture exactly the number of CbNeed evaluation steps.

Perhaps surprisingly, there is an interest in the theory of the dual *call-by-silly evaluation strategy*—which has however never been studied—that adopts both *silly duplication* and *silly erasure*, and completes the following diagram of strategies with its top corner:



The CbSilly strategy is of no practical interest for programming purposes. It is theoretically interesting, however, because it provides an *unshared*, or *inefficient* version of CbV. Let us explain what we mean:

- The kind of erasing behavior (wise/silly) determines the *termination behavior*—thus it changes the associated *contextual equivalence* and the set of *meaningful terms*.
- The kind of duplicating behavior (wise = with sharing, silly = without sharing) only affects the *efficiency* of the evaluation process, leaving contextual equivalence and meaningful terms unchanged.

Since the difference between CbN and CbNeed concerns duplications, they induce the same contextual equivalence, but—because of sharing—CbNeed is harder to study. In particular, as we outlined in a previous paragraph, CbNeed normal forms are actually DAGs, while CbN normal forms are tree-shaped. Now, a CbSilly strategy differs from CbV only with respect to duplication, which therein is silly rather than wise. Therefore, it would provide a *de-sharification* of CbV having tree-shaped normal forms, hopefully providing a simplified strategy to study CbV contextual equivalence, and possibly leading to an axiomatisation of CbV contextual equivalence.

Summing up, a fine understanding of λ-calculi with sharing declines into an understanding of how duplication and erasure are favored or deprecated, and of how these approaches can be combined.

Linear Logic? Logically, duplication and erasure immediately hint at Girard's *linear logic* [35], which is a framework where these operations are managed by a dedicated modality. But linear logic is not enough: it does model CbN and CbV, but *not* CbNeed, which is rather connected with *affine logic*, as briefly mentioned by Maraist et al. [47]. We thus expect the CbSilly strategy to also fall outside of linear logic.

8 Cost-(In)Sensitiveness and Full Abstraction

The equational mismatch between CbV and CbN goes beyond the fact that they do not have the same notion of meaningless term, and it crystallizes into the *full abstraction problem for pure CbV*.

In the literature, there are fully abstract natural models for extensions of CbV, for instance for CbV PCF by Honda and Yoshida [37], Abramsky and McCusker [2], and Koutavas et al. [40], for CbV PCF with higher-order state by Abramsky, Honda, and McCusker [1], and for the CbV quantum λ-calculus by Clairambault and de Visme [28]. There are however no such models for *pure CbV* (that is, untyped, effect-free, no arithmetic nor conditionals), and full abstraction is *not* preserved by restrictions nor extensions (PCF has arithmetic operations and conditionals, which can be seen as an extension, but it is typed, which is instead a restriction with respect to pure CbV). Moreover, there are fully abstract models for pure CbN such as the CbN relational model, see Manzonetto [45], but their CbV variant is *not fully abstract* for pure CbV, see Accattoli et al. [15].

On the positive side, it is known that *applicative bisimilarity* is fully abstract for pure CbV (see Egidi et al. [32] and Pitts [53]), but applicative bisimilarity is hard to manipulate, because its definition *quantifies over contexts*, it only uses a restricted class of contexts with respect to contextual equivalence.

Let us now look at this puzzling fact adopting a sharing-based perspective. In CbV, in fact, it is not so obvious that contextual equivalence by value, noted $\simeq_C^v$, should be the standard of reference, at least in the pure setting. The reason is that $\simeq_C^v$ is *cost-insensitive*: it equates terms such as $(\lambda x.yxx)t$ and ytt, for *any* t, also for terms t that are not values. This is against the very idea of CbV (and of sharing), of avoiding duplicating t before having evaluated it. In richer CbV settings (with non-determinism, memory locations, or probabilities), contexts discriminate more, and those terms are separated, but in the pure case they are not. The cost-insensitiveness of CbV contextual equivalence $\simeq_C^v$ suggests that $\simeq_C^v$ is actually better understood as the equivalence of the *CbSilly strategy* mentioned above. Cost-(in)sensitiveness suggests a possible explanation for the lack of full abstraction of the CbV relational model, which is that the relational model is *cost-sensitive* while $\simeq_C^v$ is *cost-insensitive*. It is likely that there are also other reasons why such a model is not fully abstract. We believe, however, that studying the question by comparing with CbSilly can help to better understand the problem.

CbNeed Models. A related point is how to define what is a denotational model of CbNeed. There is in fact no general notion of model for CbNeed. The question is delicate. As already mentioned, CbN and CbNeed contextual equivalences coincide, they only differ in the efficiency of evaluation. Roughly, this means that every model for CbN is a model for CbNeed, since models represent evaluation as an equality, and so—at first sight—the difference in efficiency cannot be observed at the denotational level.

It is however possible to establish a link between the operation of composing the interpretations $[\![t]\!]$ and $[\![s]\!]$ of t and s in the model and the evaluation of ts, as done for instance in CbN for game models by Danos et al. [30] or for the relational model by de Carvalho [31]. Guided by this intuition, it would be interesting to develop a categorical notion of CbNeed model which should *not* include all models of CbN. In particular, it should capture the CbNeed relational model of Accattoli et al. [19], which is not a model of CbN—and the only known model of CbNeed which is not a model of CbN—but not the CbN relational model. In other words, full abstraction cannot be the golden standard for CbNeed models: a new semantic principle should be isolated, which

should also allow one to distinguish between models of CbV and CbSilly (that also share the same notion of contextual equivalence).

9 Conclusions

This essay attempts to give a non-technical introduction to sharing in the λ-calculus. It hopefully provides enough evidence that the theory of the λ-calculus is *not as solid as one might expect* when sharing is added to the picture. The addition indeed raises currently open questions, forcing the development of new tools and of a new semantic theory of the λ-calculus. What is missing is a solid theoretical perspective on sharing, a holistic view encompassing operational, denotational, equational, as well as efficiency-related aspects, applicable to various evaluation schemes (call-by-name/value/need/silly, weak/strong evaluation, closed/open terms), able to modularly survive extensions with effects and further programming features, and paired with logical principles typical of Curry-Howard correspondences.

References

[1] Samson Abramsky, Kohei Honda, and Guy McCusker. 1998. A Fully Abstract Game Semantics for General References. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*. IEEE Computer Society, 334–344. https://doi.org/10.1109/LICS.1998.705669

[2] Samson Abramsky and Guy McCusker. 1997. Call-by-Value Games. In *Computer Science Logic, 11th International Workshop, CSL '97, Annual Conference of the EACSL, Aarhus, Denmark, August 23-29, 1997, Selected Papers (Lecture Notes in Computer Science, Vol. 1414)*, Mogens Nielsen and Wolfgang Thomas (Eds.). Springer, 1–17. https://doi.org/10.1007/BFb0028004

[3] Samson Abramsky and C.-H. Luke Ong. 1993. Full Abstraction in the Lazy Lambda Calculus. *Inf. Comput.* 105, 2 (1993), 159–267. https://doi.org/10.1006/inco.1993.1044

[4] Beniamino Accattoli. 2012. An Abstract Factorization Theorem for Explicit Substitutions. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan (LIPIcs, Vol. 15)*, Ashish Tiwari (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6–21. https://doi.org/10.4230/LIPIcs.RTA.2012.6

[5] Beniamino Accattoli. 2015. Proof nets and the call-by-value λ-calculus. *Theor. Comput. Sci.* 606 (2015), 2–24. https://doi.org/10.1016/j.tcs.2015.08.006

[6] Beniamino Accattoli. 2018. Proof Nets and the Linear Substitution Calculus. In *Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11187)*, Bernd Fischer and Tarmo Uustalu (Eds.). Springer, 37–61. https://doi.org/10.1007/978-3-030-02508-3_3

[7] Beniamino Accattoli. 2019. A Fresh Look at the lambda-Calculus (Invited Talk). In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany (LIPIcs, Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:20. https://doi.org/10.4230/LIPIcs.FSCD.2019.1

[8] Beniamino Accattoli. 2022. Exponentials as Substitutions and the Cost of Cut Elimination in Linear Logic. In *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, Christel Baier and Dana Fisman (Eds.). ACM, 49:1–49:15. https://doi.org/10.1145/3531130.3532445

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

[10] Beniamino Accattoli, Eduardo Bonelli, Delia Kesner, and Carlos Lombardi. 2014. A nonstandard standardization theorem. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 659–670. https://doi.org/10.1145/2535838.2535886

[11] Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. 2021. Strong Call-by-Value is Reasonable, Implosively. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–14. https://doi.org/10.1109/LICS52264.2021.9470630

[12] Beniamino Accattoli, Andrea Condoluci, Giulio Guerrieri, and Claudio Sacerdoti Coen. 2019. Crumbling Abstract Machines. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, 4:1–4:15. https://doi.org/10.1145/3354166.3354169

[13] Beniamino Accattoli and Ugo Dal Lago. 2014. Beta reduction is invariant, indeed. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 8:1–8:10. https://doi.org/10.1145/2603088.2603105

[14] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. 2022. Reasonable Space for the λ-Calculus, Logarithmically. In *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, Christel Baier and Dana Fisman (Eds.). ACM, 47:1–47:13. https://doi.org/10.1145/3531130.3533362

[15] Beniamino Accattoli, Claudia Faggian, and Adrienne Lancelot. 2023. Normal Form Bisimulations By Value. *CoRR* abs/2303.08161 (2023). https://doi.org/10.48550/arXiv.2303.08161 arXiv:2303.08161

[16] Beniamino Accattoli and Giulio Guerrieri. 2016. Open Call-by-Value. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10017)*, Atsushi Igarashi (Ed.). 206–226. https://doi.org/10.1007/978-3-319-47958-3_12

[17] Beniamino Accattoli and Giulio Guerrieri. 2019. Abstract machines for Open Call-by-Value. *Sci. Comput. Program.* 184 (2019). https://doi.org/10.1016/j.scico.2019.03.002

[18] Beniamino Accattoli and Giulio Guerrieri. 2022. The theory of call-by-value solvability. *Proc. ACM Program. Lang.* 6, ICFP (2022), 855–885. https://doi.org/10.1145/3547652

[19] Beniamino Accattoli, Giulio Guerrieri, and Maico Leberle. 2019. Types by Need. In *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 (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 410–439. https://doi.org/10.1007/978-3-030-17184-1_15

[20] Beniamino Accattoli and Delia Kesner. 2010. The Structural *lambda*-Calculus. In *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6247)*, Anuj Dawar and Helmut Veith (Eds.). Springer, 381–395. https://doi.org/10.1007/978-3-642-15205-4_30

[21] Beniamino Accattoli and Maico Leberle. 2022. Useful Open Call-By-Need. In *30th EACSL Annual Conference on Computer Science*

Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference) (LIPIcs, Vol. 216), Florin Manea and Alex Simpson (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:21. https://doi.org/10.4230/LIPIcs.CSL.2022.4

[22] Beniamino Accattoli and Luca Paolini. 2012. Call-by-Value Solvability, Revisited. In *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7294)*, Tom Schrijvers and Peter Thiemann (Eds.). Springer, 4–16. https://doi.org/10.1007/978-3-642-29822-6_4

[23] Zena M. Ariola and Matthias Felleisen. 1997. The Call-By-Need lambda Calculus. *J. Funct. Program.* 7, 3 (1997), 265–301. https://doi.org/10.1017/s0956796897002724

[24] Thibaut Balabonski, Pablo Barenbaum, Eduardo Bonelli, and Delia Kesner. 2017. Foundations of strong call by need. *Proc. ACM Program. Lang.* 1, ICFP (2017), 20:1–20:29. https://doi.org/10.1145/3110264

[25] Hendrik Pieter Barendregt. 1985. *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.

[26] Guy E. Blelloch and John Greiner. 1995. Parallelism in Sequential Functional Languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, John Williams (Ed.). ACM, 226–237. https://doi.org/10.1145/224164.224210

[27] Alberto Carraro and Giulio Guerrieri. 2014. A Semantical and Operational Account of Call-by-Value Solvability. In *Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8412)*, Anca Muscholl (Ed.). Springer, 103–118. https://doi.org/10.1007/978-3-642-54830-7_7

[28] Pierre Clairambault and Marc de Visme. 2020. Full abstraction for the quantum lambda-calculus. *Proc. ACM Program. Lang.* 4, POPL (2020), 63:1–63:28. https://doi.org/10.1145/3371131

[29] Pierre-Louis Curien and Hugo Herbelin. 2000. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 233–243. https://doi.org/10.1145/351240.351262

[30] Vincent Danos, Hugo Herbelin, and Laurent Regnier. 1996. Game Semantics & Abstract Machines. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*. IEEE Computer Society, 394–405. https://doi.org/10.1109/LICS.1996.561456

[31] Daniel de Carvalho. 2018. Execution time of λ-terms via denotational semantics and intersection types. *Math. Struct. Comput. Sci.* 28, 7 (2018), 1169–1203. https://doi.org/10.1017/S0960129516000396

[32] Lavinia Egidi, Furio Honsell, and Simona Ronchi Della Rocca. 1992. Operational, denotational and logical descriptions: a case study. *Fundam. Inform.* 16, 1 (1992), 149–169.

[33] Thomas Ehrhard. 2012. Collapsing non-idempotent intersection types. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs, Vol. 16)*, Patrick Cégielski and Arnaud Durand (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 259–273. https://doi.org/10.4230/LIPIcs.CSL.2012.259

[34] Thomas Ehrhard and Giulio Guerrieri. 2016. The Bang Calculus: an untyped lambda-calculus generalizing call-by-name and call-by-value. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, James Cheney and Germán Vidal (Eds.). ACM, 174–187. https://doi.org/10.1145/2967973.2968608

[35] Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

[36] Timothy Griffin. 1990. A Formulae-as-Types Notion of Control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, Frances E. Allen (Ed.). ACM Press, 47–58. https://doi.org/10.1145/96709.96714

[37] Kohei Honda and Nobuko Yoshida. 1999. Game-Theoretic Analysis of Call-by-Value Computation. *Theor. Comput. Sci.* 221, 1-2 (1999), 393–456. https://doi.org/10.1016/S0304-3975(99)00039-0

[38] Delia Kesner. 2016. Reasoning About Call-by-need by Means of Types. In *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 (Lecture Notes in Computer Science, Vol. 9634)*, Bart Jacobs and Christof Löding (Eds.). Springer, 424–441. https://doi.org/10.1007/978-3-662-49630-5_25

[39] Delia Kesner, Loïc Peyrot, and Daniel Ventura. 2021. The Spirit of Node Replication. In *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12650)*, Stefan Kiefer and Christine Tasson (Eds.). Springer, 344–364. https://doi.org/10.1007/978-3-030-71995-1_18

[40] Vasileios Koutavas, Yu-Yang Lin, and Nikos Tzevelekos. 2023. Fully Abstract Normal Form Bisimulation for Call-by-Value PCF. In *LICS*. 1–13. https://doi.org/10.1109/LICS56636.2023.10175778

[41] John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, Mary S. Van Deusen and Bernard Lang (Eds.). ACM Press, 144–154. https://doi.org/10.1145/158511.158618

[42] Olivier Laurent. 2002. *Étude de la polarisation en logique*. Thèse de Doctorat. Université Aix-Marseille II.

[43] Paul Blain Levy. 2006. Call-by-push-value: Decomposing call-by-value and call-by-name. *High. Order Symb. Comput.* 19, 4 (2006), 377–414. https://doi.org/10.1007/s10990-006-0480-6

[44] Giuseppe Longo. 1983. Set-theoretical models of λ-calculus: theories, expansions, isomorphisms. *Ann. Pure Appl. Log.* 24, 2 (1983), 153–188. https://doi.org/10.1016/0168-0072(83)90030-1

[45] Giulio Manzonetto. 2009. A General Class of Models of H^*. In *Mathematical Foundations of Computer Science 2009, 34th International Symposium, MFCS 2009, Novy Smokovec, High Tatras, Slovakia, August 24-28, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5734)*, Rastislav Královic and Damian Niwinski (Eds.). Springer, 574–586. https://doi.org/10.1007/978-3-642-03816-7_49

[46] Giulio Manzonetto, Michele Pagani, and Simona Ronchi Della Rocca. 2019. New Semantical Insights Into Call-by-Value λ-Calculus. *Fundam. Informaticae* 170, 1-3 (2019), 241–265. https://doi.org/10.3233/FI-2019-1862

[47] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. 1999. Call-by-name, Call-by-value, Call-by-need and the Linear lambda Calculus. *Theor. Comput. Sci.* 228, 1-2 (1999), 175–210. https://doi.org/10.1016/S0304-3975(98)00358-2

[48] John Maraist, Martin Odersky, and Philip Wadler. 1998. The Call-by-Need Lambda Calculus. *J. Funct. Program.* 8, 3 (1998), 275–317. https://doi.org/10.1017/s0956796898003037

[49] Paul-André Melliès. 1995. Typed lambda-calculi with explicit substitutions may not terminate. In *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 902)*, Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin (Eds.). Springer, 328–334. https://doi.org/10.1007/BFb0014062

[50] Robin Milner. 2006. Local Bigraphs and Confluence: Two Conjectures: (Extended Abstract). In *Proceedings of the 13th International Workshop on Expressiveness in Concurrency, EXPRESS 2006, Bonn, Germany, August 26, 2006 (Electronic Notes in Theoretical Computer Science, Vol. 175)*, Roberto M. Amadio and Iain Phillips (Eds.). Elsevier, 65–73. https://doi.org/10.1016/j.entcs.2006.07.035

[51] James Hiram Morris. 1968. *Lambda-calculus Models of Programming Languages*. Ph. D. Dissertation. MIT.

[52] Reiji Nakajima. 1975. Infinite normal forms for the lambda - calculus. In *Lambda-Calculus and Computer Science Theory, Proceedings of the Symposium Held in Rome, Italy, March 25-27, 1975 (Lecture Notes in Computer Science, Vol. 37)*, Corrado Böhm (Ed.). Springer, 62–82. https://doi.org/10.1007/BFb0029519

[53] Andrew M. Pitts. 2012. Howe's method for higher-order languages. In *Advanced Topics in Bisimulation and Coinduction*, Davide Sangiorgi and Jan J. M. M. Rutten (Eds.). Cambridge tracts in theoretical computer science, Vol. 52. Cambridge University Press, 197–232. https://doi.org/10.1017/CBO9780511792588.006

[54] Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

[55] Peter Sestoft. 1997. Deriving a Lazy Abstract Machine. *J. Funct. Program.* 7, 3 (1997), 231–264. https://doi.org/10.1017/s0956796897002712

[56] Christopher P. Wadsworth. 1971. *Semantics and pragmatics of the lambda-calculus*. PhD Thesis. Oxford.

Received 2023-04-28; accepted 2023-08-11