A Fresh Look at the λ-Calculus

Beniamino Accattoli

A Fresh Look at the λ-Calculus

Beniamino Accattoli
Inria & LIX, École Polytechnique, UMR 7161, France
beniamino.accattoli@inria.fr

Abstract

The (untyped) λ-calculus is almost 90 years old. And yet – we argue here – its study is far from being over. The paper is a bird's eye view of the questions the author worked on in the last few years: how to measure the complexity of λ-terms, how to decompose their evaluation, how to implement it, and how all this varies according to the evaluation strategy. The paper aims at inducing a new way of looking at an old topic, focussing on high-level issues and perspectives.

2012 ACM Subject Classification Theory of computation → Lambda calculus; Software and its engineering → Functional languages; Theory of computation → Operational semantics

Keywords and phrases λ-calculus, sharing, abstract machines, type systems, rewriting

Digital Object Identifier 10.4230/LIPIcs.FSCD.2019.1

Category Invited Talk

Funding *Beniamino Accattoli*: This work has been partially funded by the ANR JCJC grant COCA HOLA (ANR-16-CE40-004-01).

Acknowledgements To Herman Geuvers, Giulio Guerrieri, and Gabriel Scherer for comments on a first draft.

1 Introduction

The λ-calculus is as old as computer science. Many programming languages incorporate its key ideas, many proof assistants are built on it, and various research fields – such as proof theory, category theory, or linguistics – use it as a tool. Books are written about it, it is taught in most curricula on theoretical computer science, and it is the basis of many fashionable trends in programming languages such as probabilistic or quantum programming, or gradual type systems. Is there anything left to say about it? The aim of this informal paper is to provide evidence that yes, the theory of λ-calculus is less understood and stable than it may seem and there still are things left to say about it.

A first point is the solution of the schism between Turing machines and the λ-calculus as models of computation. The schism happened mostly to the detriment of the λ-calculus, that became a *niche* model. Despite belonging to computer science, indeed, the whole theory of the λ-calculus was developed paying no attention to cost issues. Adopting a complexity-aware point of view sheds a new light on old topics and poses new fundamental questions.

A second point is the fact that *the* λ-calculus does not exist. Despite books such as Barendregt's [31] and Krivine's [54], devoted to *the* λ-calculus, it is nowadays clear that, even if one sticks to the untyped λ-calculus with no additional features, there are a number of λ-calcu*li* depending at least on whether evaluation is call-by-name, call-by-value, or call-by-need, and – orthogonally – whether evaluation is strong or weak (that is, whether abstraction bodies are evaluated or not) and, when it is weak, whether terms are closed or can also be open. These choices affect the theory and impact considerably on the design of abstract machines. A comparative study of the various dialects allows to identify principles and differences, and to develop a deeper understanding of higher-order computations.

A third point is about decomposing λ-calculi and the role of sharing. More or less ubiquitously, for practical as well as for theoretical reasons, λ-calculi are presented with `let` expressions, that are nothing else but constructs for first-class sharing, logically corresponding to *the* logical rule behind computation, the *cut* rule. Sharing is also essential for the two previous points, complexity-aware understandings of λ-calculi and defining evaluation strategies such as call-by-need. Perhaps surprisingly, there are simple extensions with sharing that have important properties that "the" λ-calculus lacks. Therefore, there is a third *sharing axis* – beyond the strong/weak axis and the call-by-name/value/need axis – along which the λ-calculus splits into various calculi, that should gain the same relevant status of the other λ-calculi.

This paper tries to explain and motivate these points, through an informal overview of the author's research. The unifying theme is the ambition to harmonise together theoretical topics such as denotational semantics or advanced rewriting properties and practical studies such as abstract machines and call-by-need, using as key ingredients sharing and cost analyses.

2 The Higher-Order Time Problem

The *higher-order time problem* is the problem of how to measure the time complexity of programs expressed as λ-terms. It is a subtle and neglected problem. There are indeed no traces of the question in the two classic books on the λ-calculus, Barendegt's and Krivine's, for instance.

Size Explosion. The λ-calculus has only one computational rule, β-reduction, and the question is whether it is possible to take the number of β-steps as a reasonable time measure or not. Of course, the question is subtler than that, because a λ-term in general admits many different evaluation sequences, of different length, so that one should rather first fix an evaluation strategy. Unfortunately, we have to first deal with a deeper difficulty that affects every strategy.

The essence of the problem indeed amounts to a degeneracy, called *size explosion*: there are families of programs $\{t_n\}_{n\in\mathbb{N}}$ such that t_n evaluates in n β steps to a result s_n of size exponential in n (and in the size $|t_n|$ of t_n). Now, it seems that the number of β-steps cannot be a reasonable measure, because n – the candidate measure of time complexity – does not even account for the time to write down the exponential result s_n.

Size explosion is extremely robust: there is an exploding family $\{t_n\}_{n\in\mathbb{N}}$ (for details see [6]) such that

- t_n is closed;
- all its evaluation sequences to normal form have the same length;
- strong and weak evaluations produce the same result, in the same number of steps;
- lives in the continuation-passing style fragment of the λ-calculus, what is sometimes considered as a sort of simpler kernel;
- can even be typed with simple types.

Moreover, even if one restricts to, say, terms evaluating to booleans, for which normal forms have constant size, size explosion can still happen *along the way*: evaluation can produce a sub-term of exponential size and then erase it, making the number of steps a doubtful notion of cost model even if it is no longer possible to produce an exponential normal form.

Phrased differently, there are no easy tricks: size explosion is here to stay.

Surprisingly, until very recently size explosion was folklore. There are no traces of it in the major literature on the λ-calculus. It was known that duplications in the λ-calculus can have a nasty behaviour, and this is why the λ-calculus is never implemented as it is defined –

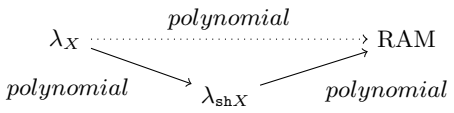
all implementations indeed rely on some form of sharing. In 2006, Dal Lago and Martini started the first conscious exploration of the higher-order time problem [39], having instances of size explosion in mind. But it is only in 2014 that Accattoli and Dal Lago named the degeneracy *size explosion* [17], taking it out from the collective unconscious, and starting a systematic exploration.

Reasonable Cost Models. What does it exactly mean for a cost measure to be reasonable? First of all, one has to fix a computational model M, whose role in our case is played by the λ-calculus. As proposed by Slot and van Emde Boas in 1984 [88], a time cost model for M is *reasonable* when there are simulations of M by Turing machines and of Turing machines by M having overhead bounded by a polynomial of the input and of the number of steps (in the source model). For space, similarly, one requires a linear overhead. The basic idea is to preserve the complexity class ¶, that then becomes *robust*, that is, model-independent. Random access machines for instance are a reasonable model (if multiplication is a primitive operation, they need to be endowed with a logarithmic cost model).

The question becomes: are there reasonable cost models for the λ-calculus? At least for time? In principle, everything can be a cost model, but of course one is mainly interested in the natural one, the number of β-steps. The precise question then is: are there reasonable evaluation strategies, that is, strategies whose number of steps is a reasonable time cost model?

Sharing. The good news is that there are strategies for which size explosion is avoidable, or, rather, it can be circumvented, and so, the answer is yes, reasonable strategies do exist. The price to pay is the shift to a λ-calculus enriched with *sharing* of subterms. Roughly, size explosion is based on the blind duplication of useless sub-terms – by keeping these sub-terms shared, the exponential blow up of the size can be avoided and the number of β-steps can be taken as a reasonable measure of time complexity.

Fix a dialect λ_X of the λ-calculus with a deterministic evaluation strategy $\to_X$, and note $\mathtt{nf}_X(t)$ the normal form of t with respect to $\to_X$. The idea is to introduce an intermediate setting $\lambda_{\mathtt{sh}X}$ where λ_X is refined with sharing (we are vague about sharing on purpose) and evaluation in λ_X is simulated by some refinement $\to_{\mathtt{sh}X}$ of $\to_X$. The situation can then be refined as in the following diagram:



(1)

In the best cases [85, 13, 21] the simulations have bilinear overhead, that is, linear in the number of steps and in the size of the initial term. A term with sharing t represents the ordinary term $t{\downarrow}$ obtained by unfolding the sharing in t – the key point is that t can be exponentially smaller than $t{\downarrow}$. Evaluation in $\lambda_{\mathtt{sh}X}$ produces a shared normal form $\mathtt{nf}_{\mathtt{sh}X}(t)$ that is a compact representation of the ordinary result, that is, such that $\mathtt{nf}_{\mathtt{sh}X}(t){\downarrow} = \mathtt{nf}_X(t)$.

Let us stress that one needs sharing to obtain the simulations but then the strategy proved to be reasonable is the one without sharing (that is, $\to_X$) – here sharing is the key tool for the proof, but the cost model is not taken on the calculus with sharing.

Sharing and Reasonable Strategies. The kind of sharing at work in diagram (1), and therefore the definitions of $\lambda_{\mathtt{sh}X}$ and $\to_{\mathtt{sh}X}$, depends very much on the strategy $\to_X$. Let us fix some terminology.

The *weak* λ-calculus is the sub-calculus in which evaluation does not enter into abstractions, and, with the additional hypothesis that terms are *closed*, it models functional programming languages. The strong λ-calculus is the case where evaluation enters into function bodies, and its main domain of application are proof assistants.

The first result for weak strategies is due to Blelloch and Greiner in 1995 [32] and concerns weak CbV evaluation. Similar results were then proved again, independently, by Sands, Gustavsson, and Moran in 2002 [85] who also addressed CbN and CbNeed, and by combining the results by Dal Lago and Martini in 2009 in [41] and [40], who also addressed CbN in [61]. Actually already in 2006, Dal Lago and Martini proposed a reasonable cost model for the CbV case [39], but that cost model does not count 1 for each β-step.

Note, *en passant*, that *reasonable* does not mean *efficient*: CbN and CbV are incomparable for efficiency, and CbNeed is more efficient than CbN, and yet they are all reasonable. *reasonable* and *efficient* are indeed unrelated properties of strategies. Roughly, efficiency is a *comparative* property, it makes sense only if there are many strategies and one aims at comparing them. Being reasonable instead is a property of the strategy itself, independently of any other strategy, and it boils down to the fact that the strategy can be implemented with a negligible overhead.

In the strong case, at present, only one reasonable strategy is known, the leftmost-outermost (LO) strategy, that is the extension of the CbN strategy to the strong case. The result is due to Accattoli and Dal Lago [17] (2014), and it is inherently harder than the weak case, as a more sophisticated notion of sharing is required.

There also is an example of an unreasonable strategy. Asperti and Mairson in [27] (1998) proved that Lévy's parallel optimal evaluation [67] is unreasonable. Careful again: unreasonable does not mean inefficient (but it does not mean efficient either).

An Anecdote. To give an idea of the subtlety but also of how much these questions are neglected by the community, let us report an anecdote. In 2011, we attended a talk where the speaker started by motivating the study of strong evaluation as follows. There exists a family $\{s_n\}_{n\in\mathbb{N}}$ of terms such that s_n evaluates in $\Omega(2^n)$ steps with any weak strategy while it takes $\mathcal{O}(n)$ steps with rightmost-innermost strong evaluation. Thus, the speaker concluded, strong evaluation can be faster than weak evaluation, and it is worth studying it.

Such a reasoning is wrong (but the conclusion is correct, it is worth studying strong evaluation!). It is based on the hidden assumption that it makes sense to count the number of β-steps and compare strategies accordingly. Such an assumption, however, is valid only if the compared strategies are reasonable, that is, if it is proved that their number of steps is a reasonable time measure. In the talk, the speaker was comparing weak strategies, of which we have various reasonable examples, together with a strong strategy that is not known to be reasonable. In particular, rightmost-innermost evaluation is probably unreasonable, given that even when decomposed with sharing it lacks one of the key properties (the sub-term property) used in all proofs of reasonability in the literature.

That talk was given in front of an impressive audience, including many big names and historical figures of the λ-calculus. And yet no one noticed that identifying the number of β steps with the actual cost was naive and improper.

Apart from avoiding traps as the one of the anecdote, a proper approach to the cost of computation sheds a new light on questions of various nature. The next two subsections discuss the practical case of abstract machines and the theoretical case of denotational semantics.

2.1 Abstract Machines

The first natural research direction is the re-understanding of implementation techniques from a quantitative point of view.

Environment-Based Abstract Machines. The theory of implementations of λ-calculi is mainly based on environment-based abstract machines, that use *environments* to implement sharing and avoid size explosion. Having a reasonable cost model of reference, namely the number of β-steps of the strategy implemented by the machine, it is possible to bound the complexity of the machine overhead as a function of the size of the initial term and the number of steps taken by the strategy in the calculus.

A complexity-based approach to abstract machines is not a pedantic formality: Cregut's machine [37], that was the only known machine for strong evaluation for 25 years, has an exponential overhead with respect to the number of β-steps. Essentially, Cregut machine is as bad as implementing β-reduction *literally* as it is defined, without any form of sharing.

Similarly, the abstract machine for open terms described by Grégoire and Leroy in [50] suffers of exponential overhead, even if the authors in practice implement a slightly different machine with polynomial overhead.

In collaboration with Barenbaum, Mazza, Sacerdoti Coen, Guerrieri, Barras, and Condoluci, we developed a new theory of abstract machines [9, 10, 13, 7, 14, 11, 21, 15], where different term representations, data structures, and evaluation techniques are studied from a complexity point of view, compared, and sometimes improved to match a certain complexity. Some of the outcomes have been:

- A reasonable variant of Cregut's machine [7].
- A detailed study of how to improve Grégoire and Leroy's machine [13, 21].
- The first results showing that de Bruijn indices bring no asymptotic speed-up with respect to using names and perform α-renaming [11].
- The proof that administrative normal forms bring no asymptotic slowdown [15] (in contrast to what claimed by Kennedy in [52]).
- and, as a side contribution, the simplest presentations of CbNeed [9].

Apart from two exceptions actually focusing on other topics (the already cited studies by Blelloch and Greiner [32] and by Sands, Gustavsson, and Moran [85]), the literature before this new wave never studied the complexity of abstract machines.

Token-Based Abstract Machines. There is another class of abstract machines that is much less famous than those based on environments. These are so-called *token-based* abstract machines, introduced by Danos and Regnier [43], then studied by Mackie [69], Schöpp [86] and more recently by Mazza [72], Muroya and Ghica [78], and Dal Lago and coauthors [56, 57, 62, 58]. Their theoretical background is Girard's geometry of interaction. The basic idea is that, instead of storing all previously encountered β-redexes in environments, these machines keep a minimalistic amount of information inside a data structure called *token*, that is used to navigate the program without ever modifying it. The aim is to have an execution model that sacrifices time, by possibly repeating some work already done, in order to be efficient with respect to space. Various researchers conjecture the size of the token to be a reasonable cost model for space, but there are no results in the literature.

2.2 Denotational Semantics

Another research direction is the connection between the cost of computations and denotational semantics. At first sight, it looks like a non-topic, because denotational semantics are invariant under evaluation, and so it seems that they cannot be sensitive to intensional properties such as evaluation lengths. There are however hints that the question is subtler than it seems at first sight.

There are works in the literature trying to address somehow the question, by either building model of logics / λ-calculi with bounded complexity [77, 28, 59, 66], or by designing resource-sensitive models [48, 60, 63], or by extracting evaluation bounds from some semantics, typically game semantics [34, 35, 26]. The questions we propose here are related, but – at present – very open and somewhat vague. They stem from the work of Daniel de Carvalho on non-idempotent intersection types [44] (finally published in 2018 but first appeared in 2007), or, as we prefer to call them, *multi types* (because non-idempotent intersections can be seen as multi-sets).

One of de Carvalho's ideas is that even if the interpretation $[\![t]\!]$ of a λ-term t cannot tell us anything about the evaluation of t itself, there is still hope that (when t and s are normal) $[\![t]\!]$ and $[\![s]\!]$ provide information about the evaluation of ts, typically about the number of steps to normal form and the size of the normal form. De Carvalho studies the CbN relational model (induced by the relational model of linear logic via the call-by-name translation of λ-calculus into linear logic), that can be syntactically presented via multi types. The key points of his study are:

1. the size of a type derivation π of $\Gamma \vdash t : A$ provides bounds to both the number of CbN steps to evaluate t to its normal form $\mathtt{nf}(t)$ and the size $|\mathtt{nf}(t)|$ of $\mathtt{nf}(t)$.
2. the size of the types themselves – more precisely A plus the types in the typing context Γ – bounds $|\mathtt{nf}(t)|$.
3. the interpretation of a term in the model is the set of type judgements – again, A plus the types in the typing context Γ – that can be derived for it in the typing system, that is,

$$[\![t]\!] := \{((M_1, \dots, M_n), A) \mid x_1 : M_1, \dots, x_n : M_n \vdash t : A\}\,.$$

   where the M_i are multi-sets of types.
4. Minimal derivations provide the exact measure of the number of steps *plus* the size of the normal form. Similarly the minimal derivable types provide the exact measure of the normal form.
5. From $[\![t]\!]$ and $[\![s]\!]$ one can bound the number of steps *plus* the size of the normal form of ts.

Such a strong correspondence does not happen by chance. Further work [45, 19] has shown that multi types and the relational model compute according to natural strategies in linear logic proof nets, including in particular CbN evaluation. The link is natural: multi types are intersection types without idempotency, that is, without the principle $A \cap A = A$, or, said differently, the number of times that A is appear does matter... exactly as in linear logic. Similar results connect strategies in linear logic proof nets and game semantics [42, 34, 35].

The Extraction of Computational Mechanisms from Models. De Carvalho's work suggests that denotational models hide a computational machinery behind their compositional principles. The one between relational and game semantics and evaluation strategies may be only the easiest one to observe. A natural question is: what about (idempotent) intersection types? They are syntactic presentations of domain-based semantics *à la Scott*. Despite being the first discovered model of the λ-calculus, nothing is known about their hidden evaluation

scheme, apart from the fact that it is not the one behind the relational model, for which non-idempotency is required. Intuition says that idempotency may model some form of sharing. The question is however open.

Models Internalising Sharing. The key points about size explosion and reasonable cost models are:

1. In an evaluation to normal form $t \to_\beta^n \mathtt{nf}(t)$ the size $|\mathtt{nf}(t)|$ of the normal form may be exponential in n;
2. n is nonetheless a reasonable measure of complexity (if evaluation is done according to a reasonable strategy);
3. this is possible because sharing allows to compute a compact representation $\mathtt{nf}_{\mathtt{sh}X}(t)$ of the normal form that is polynomial in n.

Now, the interpretation of a term t in de Carvalho's CbN relational model is a set whose smallest element is as large as the normal form $\mathtt{nf}(t)$. It seems then difficult that such a model may give accurate information about the time cost of λ-terms, as in general from $[\![t]\!]$ and $[\![s]\!]$ one can only obtain information about the evaluation of ts *together* with the size of $\mathtt{nf}(ts)$, which may however be much larger than the length of evaluation to normal form.

For such a reason, Accattoli, Graham-Lengrand, and Kesner in [19] refined de Carvalho's type system as to have type derivations that provide separate bounds with respect to evaluation lengths and the size of normal forms. The idea is that judgements now have the form:

$$x_1 : M_1, \dots, x_n : M_n \vdash^{(b,r)} t : A$$

where b provides a bound to number of β-steps to evaluate t to its normal form $\mathtt{nf}(t)$ (that is, the evaluation length) and r is a bound to the size of $\mathtt{nf}(t)$. This is a slight improvement, but still not enough, as such a refined information is on type derivations but not on the types themselves, that are what defines the semantical interpretation.

An important open question is then whether there are models where the (smallest point in the) interpretation of t is of the order of the size of the compact representation $\mathtt{nf}_{\mathtt{sh}X}(t)$ of the normal form, and not of the order of $|\mathtt{nf}(t)|$. Roughly, it would be a model whose hidden computational mechanism uses sharing as it is needed to obtain reasonable implementations.

We believe that this is an important question to answer in order to establish a semantical understanding of sharing and close the gap between semantical and syntactical studies.

We conjecture that the CbV relational model may have this property, as – despite being built from non-idempotent intersection types – it allows a special use of the empty intersection, which is the only idempotent type of the model (as the intersection of two empty intersections is still empty) and that may be the key tool to internalise sharing.

Lax and Tight Models with Respect to Strategies. A related question is how to refine the notion of model as to be relative to an evaluation strategy. The need for a refined notion of model arises naturally when studying CbNeed evaluation. CbNeed is sometimes considered simply as an optimisation of CbN. It is however better understood as an evaluation scheme on its own, obtained by mixing the good aspects of both CbN and CbV, and observationally equivalent to CbN. Because of such an equivalence, every model of CbN provides a model of CbNeed. In particular, the relational model built on multi types discussed above is a model of CbNeed – Kesner used it to provide a simple proof of the equivalence of CbN and CbNeed [53].

The bounds provided by that relational model, however, are not exact for CbNeed, since they are exact for CbN, and thus cannot capture its faster alternative. Recently, Accattoli, Guerrieri, and Leberle have obtained a multi type system providing exact bounds for CbNeed [23]. The type system induces a model, which is "better" than de Carvalho's one, as it more precisely captures CbNeed evaluation lengths. There is however no abstract, categorical way – at present – to separate the two models, as there are no abstract notions of lax or tight model with respect to an evaluation strategy. To be fair, there is not even a notion of categorical model of CbNeed, that should certainly be developed.

3 From the λ-calculus to λ-calculi

There are at least two theories of the λ-calculus, the strong and the weak. Historically, the theory of λ-calculus rather dealt with *strong* evaluation, and it is only since the seminal work of Abramsky and Ong [2] that the theory took weak evaluation seriously. Dually, the practice of functional languages mostly ignored *strong* evaluation, with the notable exception of Crégut [36, 37] (1990) and, more recently, the semi-strong approach of Grégoire and Leroy [50] (2002), following the idea that a function is an algorithm to apply to some data rather than data by itself. Strong evaluation is nonetheless essential in the implementation of proof assistants or higher-order logic programming, typically for type-checking in frameworks with dependent types as the Edinburgh Logical Framework or the Calculus of Constructions, as well as for unification modulo $\beta\eta$ in simply typed frameworks like λ-prolog.

There is also another axis of duplication of work. Historically, the theory is mostly studied with respect to CbN evaluation, while functional programming languages tend to employ CbV (of which there are no traces in Barendregt's and Krivine's books) or CbNeed. The differences between these settings is striking. What is considered *the* λ-calculus is the strong CbN λ-calculus, and it has been studied in depth, despite the fact that no programming language or proof assistant implements it. Given the practical relevance of weak CbV with closed terms, that is the backbone of the languages of the ML family, such a setting is also well known, even if its theory is anyway less developed than the one for strong CbN. Weak CbN is also reasonably well studied. But as soon as one steps out of these settings the situation changes drastically. The simple extension of weak CbV to open terms is already a delicate subject, not to speak of strong CbV. About CbNeed– that is the strategy implemented by Haskell – the situation is even worse, as its logical (Curry-Howard) understanding is less satisfactory than for CbN and CbV, its semantical understanding essentially inexistent, and there is only one paper about Strong CbNeed, by Balabonski et al. [29].

We believe that there is a strong need of reconciling theory and practice. A key step has to be a change of perspective. The λ-calculus comes in different flavors, and in my experience a comparative study is very fruitful, as it identifies commonalities, cleaning up concepts, and also stresses differences and peculiar traits of each setting.

The Open Setting. There actually is an intermediate setting between the weak and the strong λ-calculi. First of all, it is important to stress that weak evaluation is usually paired with the hypothesis that terms are *closed*, which is essential in order to obtain the key property that weak normal forms are all and only abstractions – this is why we rather prefer to call it the *closed λ-calculus*. On the other hand, in the strong case it does not make sense to restrict to closed terms, because evaluation enters into function bodies, that cannot be assumed closed. Such a difference forbids to see the strong case as the iteration of the closed one under abstraction, as the closed hypothesis is not stable under iterations.

It is then natural to consider the case of weak evaluation with open terms, what we like to refer to as the *open λ-calculus*. The open λ-calculus can be iterated under abstraction providing a procedure for strong evaluation – this is for instance done by Grégoire and Leroy in [50]. Historically, the open case was neglected. The reason is that the differences between the closed and open case are striking in CbV but negligible in CbN. Since the CbV literature focused on the closed setting, the open case sat in a blind spot.

The study of reasonable cost models made evident that different, increasingly sophisticated techniques are needed in the three settings – closed, open, and strong – in order to obtain reasonable abstract machines. Moreover, such a classification is stable by moving on the other axis, that is, closed CbN, closed CbV, and closed CbNeed share the same issues, and similarly for the three open cases and the three strong cases.

3.1 Open Call-by-Value

The ordinary approach to CbV, due to Plotkin, has a famous property that we like to call *harmony*: a closed term either is a value or it CbV reduces.

The Issue with Open Terms. Plotkin's operational semantics for CbV does have some good properties on open terms. For instance, it is confluent and it admits a standardisation theorem, as Plotkin himself proved [81]. It comes however also with deep problems. First, open terms bring *stuck β-redexes* such as

$$(\lambda x.t)(yz)$$

the argument is not a value and will never become one, thus the term is CbV normal (there is a β-redex but it is not a β_v-redex) – that is, harmony is lost.

Unfortunately, stuck redexes induce a further problem: they block creations. Consider the open term (where δ is the usual duplicator)

$$((\lambda x.\delta)(yz))\delta$$

As before, it is normal in Plotkin's traditional λ-calculus for CbV. Now, however, there are semantic reasons to consider it as a divergent term. Roughly, there are denotational semantics that are *adequate* on closed terms (adequacy means that the semantical interpretation of a term t is non-empty if and only if t evaluates to a value; by harmony, it is equivalent to say that t is *not* divergent) and with respect to which $(\lambda x.\delta)(yz)\delta$ has empty interpretation (that is, it is considered as being a divergent term, while it is normal). This semantical mismatch has first been pointed out by Paolini and Ronchi Della Rocca [80, 79, 82]. A similar phenomenon happens if one looks at the interpretation of the term according to the CbV translation into linear logic, as pointed out by the author in [5]. Essentially, that term is expected to reduce as follows

$$((\lambda x.\delta)(yz))\delta \to \delta\delta$$

and *create* the redex $\delta\delta$. Evaluation then would go on forever. If one sticks to Plotkin's rewriting rule, however, the creation is blocked by the stuck redex $(\lambda x.\delta)(yz)$ and the term is normal – quite a mismatch with what is expected semantically and logically. A similar problem affects the term $\delta((\lambda x.\delta)(yz))$ that is also normal while it should be divergent.

The subtlety of the problem is that one would like to have a notion of CbV evaluation on open terms making terms such as $((\lambda x.\delta)(yz))\delta$ and $\delta((\lambda x.\delta)(yz))$ divergent, thus extending Plotkin's evaluation, but at the same time preserving CbV divergence without collapsing on CbN, that is, such that $(\lambda x.y)(\delta\delta)$ has no normal form.

Open CbV. In his seminal work, Plotkin already pointed out an asymmetry between CbN and CbV: his continuation-passing style (CPS) translation is sound and complete for CbN, but only sound for CbV. This fact led to a number of studies about monad, CPS, and logical translations [76, 83, 84, 70, 46, 51] that introduced many proposals of improved calculi for CbV. The dissonance between open terms and CbV has been repeatedly pointed out and studied *per se* via various calculi related to linear logic [24, 5, 33, 13]. To improve the implementation of the Coq proof assistant, Grégoire and Leroy introduced another extension of CbV to open terms [50]. A further point of view on CbV comes from the computational interpretation of sequent calculus due to Curien and Herbelin [38]. An important point is that most of these works focus on strong CbV.

Inspired by the robustness of complexity classes via reasonable cost models, in a series of works Accattoli, Guerrieri and Sacerdoti Coen define what they call *open CbV*, that is the isolation of the case of weak evaluation with open terms (rather than strong evaluation) that they show to be a simpler abstract framework with many different incarnations:

- *Operational semantics*: in [20], 4 representative calculi extending Plotkin's CbV are compared, and showed to be termination equivalent (the evaluation of a fixed term t terminates in one of the calculi if and only if terminates in the others). Moreover, evaluation lengths (in 3 of them) are linearly related;
- *Cost model*: In [13, 21], such a common evaluation length is proved to be a reasonable cost model, by providing abstract machines that are proved reasonable;
- *Denotational semantics*: In [22], Ehrhard's relational model of CbV [47] is shown to be an adequate denotational model of open CbV, providing exact bounds along the lines of de Carvalho's work.

Last, let us stress both the termination equivalence of the 4 presentations of open CbV and the relationship with the denotational semantics make crucial use of formalisms with sharing.

A key point is that each presentation of open CbV has its pros and cons, but none of them is perfect. This is in contrast to CbN, where there are no doubts about the canonicity of its presentation.

Strong CbV. The obvious next step is to lift the obtained relationships to strong evaluation. This is a technically demanding ongoing work.

3.2 Benchmark for λ-Calculi

The work on open CbV forced to ask ourselves what are the guiding principles that define *a good λ-calculus*. This is especially important if one takes seriously the idea that there is not just one λ-calculus but a rich set of λ-calculi, in order to fix standards and allow comparisons.

It is of course impossible to give an absolute answer, because different applications value different properties. It is nonetheless possible to collect requirements that seem desirable in order to have an abstract framework that is also useful in practice. We can isolate at least six principles to be satisfied by a good λ-calculus:

1. *Rewriting*: there should be a small-step operational semantics having nice rewriting properties. Typically, the calculus should be non-deterministic but confluent, and a deterministic evaluation strategy should emerge naturally from some good rewriting property (factorisation / standardisation theorem, or the diamond property). The *strategy emerging from the calculus* principle guarantees that the chosen evaluation is not ad-hoc.
2. *Logic*: typed versions of the calculus should be in Curry-Howard correspondences with some proof systems, providing logical intuitions and guiding principles for the features of the calculus and the study of its properties.

3. *Implementation*: there should be a good understanding of how to decompose evaluation in micro-steps, that is, at the level of abstract machines, in order to guide the design of languages or proof assistants based on the calculus.
4. *Cost model*: the number of steps of the deterministic evaluation strategy should be a reasonable time cost model, so that cost analyses of λ-terms are possible and independent of implementative choices.
5. *Denotations*: there should be denotational semantics that reflect some of its properties, typically an adequate semantics reflecting termination. Well-behaved denotations guarantee that the calculus is somewhat independent from its own syntax, which is a further guarantee that it is not ad-hoc.
6. *Equality*: contextual equivalence can be characterised by some form of bisimilarity, showing that there is a robust notion of program equivalence. Program equivalence is indeed essential for studying program transformations and optimisations at work in compilers.

Finally, there is a sort of meta-principle: the more principles are connected, the better. For instance, it is desirable that evaluation in the calculus correspond to cut-elimination in some logical interpretation of the calculus. Denotations are usually at least required to be *adequate* with respect to the rewriting: the denotation of a term is non-degenerate if and only if its evaluation terminates. Additionally, denotations are *fully abstract* if they reflect contextual equivalence. And implementations have to work within an overhead that respects the intended cost semantics. Ideally, all principles are satisfied and perfectly interconnected.

Of course, some specific cases may drop some requirements – for instance, a probabilistic λ-calculus would not be confluent – some properties may also be strengthened – for instance, equality may be characterised via a separation theorem akin to Böhm's – and other principles may be added – categorical semantics, graphical representations, etc. As concrete cases, the strong CbN λ-calculus satisfies all these principles, while at present open CbV satisfies the first 5, with an high degree of connection between them, strong CbNeed only 2 or 3 of them, and strong CbV none of them.

We are here exposing these principles hoping to receive feedback from the community, for instance, helping us identifying further essential principles, if any. We also believe that single researchers tend to specialise excessively in one of the principles, forgetting the global picture, which is instead where the meaning of the single studies stems, in our opinion. A new book or new introductory notes on the λ-calculus should be developed around the idea of connecting these principles, to form students in the field.

4 Sharing

The aim of this section is to convince the reader that sharing is an unavoidable ingredient of a modern understanding of λ-calculi. This is done by pointing out a number of reasons, including a historical perspective, and giving some examples coming from the work of the author. In particular, we would like to stress that, rather than a feature such as continuations or pattern matching that can be added on top of λ-calculi, sharing is the *éminence grise* of the higher-order world.

Sharing is a vague term that means different things in different contexts. For instance, sharing as in environment-based abstract machines or sharing as in Lamping's sharing graphs [64] implementing Lévy's parallel optimal strategy [67] have not much in common.

Here we refer to the simplest possible form, that is closer to sharing as it appears in abstract machines. While we do want to commit to a certain style, as it shall be evident below, we also want to stay vague about it, as such a style can be realized in various ways.

The simplest construct for sharing is a $\texttt{let}\ x = s\ \texttt{in}\ t$ expression, that is a syntactic annotation for *t where x will be substituted by s*. We also write it more concisely as $t[x \leftarrow s]$ (not to be confused with meta-level substitution, noted $t\{x \leftarrow s\}$) and call it ES (for *explicit sharing*, or *explicit substitution*). Thanks to ES, β-reduction can be decomposed into more atomic steps. The simplest decomposition splits β-reduction as follows:

$$(\lambda x.t)s \quad \to_{\mathsf{B}} \quad t[x \leftarrow s] \quad \to_{\mathsf{ES}} \quad t\{x \leftarrow s\}$$

It is well-known that ES are somewhat redundant, as they can always be removed, by simply coding them as β-redexes. They are however more than syntactic sugar, as they provide a simple and yet remarkably effective tool to understand, implement, and program with λ-calculi and functional programming languages:

- From a logical point of view ES are the proof terms corresponding to the extension of natural deduction with a cut rule, and the cut rule is *the* rule representing computation, according to Curry-Howard.
- From an operational semantics point of view, they allow elegant formulations of subtle strategies such as call-by-need evaluation – various presentations of call-by-need use ES [89, 65, 71, 25, 87, 55] and a particularly simple one is in [9].
- From a programming point of view, `let` expressions are part of the syntax of all functional programming languages we are aware of.
- From a rewriting point of view, they enable proof techniques that are not available within the λ-calculus, as we are going to explain below.
- Finally, sharing is used in all implementations of tools based on the λ-calculus to circumvent *size explosion*.

A Historical Perspective. Between the end of the eighties and the beginning of the nineties, three independent decompositions of the λ-calculus arose with different aims and techniques:

1. Girard's *linear logic* [49], where the λ-calculus is decomposed in two layers, *multiplicative* and *exponential*;
2. Abadi, Cardelli, Curien, and Lévy's *explicit substitutions* [1], that are refinements of the λ-calculus where meta-level substitution is delayed, by introducing explicit annotations, and then computed in a micro-step fashion.
3. Milner, Parrow, and Walker's *π-calculus* [75], where the λ-calculus can be represented, as shown by Milner [73], by decomposing evaluation into message passing and process replication.

All these settings introduce an explicit treatment of sharing – called *exponentials* in linear logic, or explicit substitutions, or *replication* in the π-calculus. At first sight these approaches look quite different. It took more than 20 years to obtain the *Linear Substitution Calculus* (LSC), a λ-calculus with sharing that captures the essence of all three approaches in a simple and manageable formalism.

The LSC [3, 12] is a refinement of the λ-calculus with sharing, introduced by Accattoli and Kesner as a minor variation over a calculus by Milner [74]. The rest of the section is a gentle introduction to some of its unique features. We shall also discuss an even simpler setting, the *substitution calculus*, that is probably the most basic setting for sharing.

4.1 Rewriting at a Distance, or Disentangling Search and Substitution

Usually, λ-calculi with ES have rules accounting for two tasks, one is decomposing (or just executing) the substitution process and one is commuting ES with other constructs. For the time being, let us simplify the first task, and assume that ES are executed in just one step, as follows

$$t[x{\leftarrow}s] \quad \to_{\mathsf{ES}} \quad t\{x{\leftarrow}s\}$$

The second task is required for instance in situations such as

$$(\lambda x.t)[y{\leftarrow}s]u$$

where the ES $[y{\leftarrow}s]$ is blocking the possibility of reducing the β redex $(\lambda x.t)u$. Usually the calculus is endowed with one of the two following rules

$$(\lambda x.t)[y{\leftarrow}s] \to_{\lambda} \lambda x.t[y{\leftarrow}s] \qquad\qquad t[y{\leftarrow}s]u \to_{@} (tu)[y{\leftarrow}s]$$

that incarnate opposite approaches to expose the β-redex and continue the computation.

There is however a simpler alternative. Instead of adding a rule to commute ES, one can generalise the notion of β-redex, by allowing the abstraction and the argument to interact *at a distance*, that is, even if there are ES in between:

$$(\lambda x.t)[y_1{\leftarrow}s_1]\dots[y_k{\leftarrow}s_k]u \quad \to_{\mathsf{dB}} \quad t[x{\leftarrow}u][y_1{\leftarrow}s_1]\dots[y_k{\leftarrow}s_k]$$

The name of the rule stands for *distant* B, where B is the variant of β that creates an ES.

The rule can be made compact by employing *contexts*. Define *substitution contexts* as follows

$$L ::= \langle\cdot\rangle \mid L[x{\leftarrow}t]$$

Then the previous rule can be rewritten as:

$$L\langle\lambda x.t\rangle u \to_{\mathsf{dB}} L\langle t[x{\leftarrow}u]\rangle$$

Define the *substitution calculus* as the language of the λ-calculus with ES plus rules $\to_{\mathsf{dB}}$ and $\to_{\mathsf{ES}}$. Such a simple formalism already provides relevant insights.

First of all, it has a strong connection with the linear logic proof nets representation of the λ-calculus [8]. Rules $\to_{\mathsf{dB}}$ and $\to_{\mathsf{ES}}$ indeed correspond *exactly* to multiplicative and exponential cut-elimination in such a representation (if one assumes a *one shot* cut-elimination rule for the exponentials), where *exactly* means that there is a bijection of redexes providing a *strong* bisimulation between terms and proof nets (of that fragment). This happens because the *trick* of rewriting at a distance captures exactly the graphical dynamics of proof nets. Phrased differently, commuting rules such as $\to_{\lambda}$ or $\to_{@}$ have no analogous on proof nets.

4.2 A Rewriting Pearl: Confluence from Local Diagrams

The substitution calculus already allows to show that sharing enables simple and elegant proof techniques that are impossible in λ-calculi without sharing.

The best example concerns confluence. In rewriting, termination often allows to lift local properties to the global level. The typical example is Newman lemma, that in presence of strong normalisation lifts local confluence to confluence. The λ-calculus is locally confluent but not strongly normalising (not even weakly!), so Newman lemma cannot be applied. It is then necessary, for instance, to introduce parallel steps and adopt the Tait and Martin Löf proof technique.

In the substitution calculus, instead, we can prove confluence from local confluence. Newman lemma does not apply directly, but an elegant alternative reasoning is possible.

The key observation is that having decomposed β in two rules $\to_{\mathtt{dB}}$ and $\to_{\mathtt{ES}}$, one still has that the whole calculus may not terminate, but a new *local* termination principle is available: $\to_{\mathtt{dB}}$ and $\to_{\mathtt{ES}}$ are strongly normalising when considered separately. Local termination can be seen as the internalisation of a classic result, the *finite developments theorem*, stating that in the λ-calculus every evaluation that does not reduce redexes created by the sequence itself terminates. The proof of confluence of the substitution calculus goes as follows:

- Rules $\to_{\mathtt{dB}}$ and $\to_{\mathtt{ES}}$ are both locally confluent, so that by Newman lemma they are (separately) confluent.
- To obtain confluence of the full calculus we need another classic rewriting lemma by Hindley and Rosen: the union of confluent and commuting relations is confluent.
- We then need to prove commutation of $\to_{\mathtt{dB}}$ and $\to_{\mathtt{ES}}$, that is, if $s \,{}^{*}_{\mathtt{ES}}\!\leftarrow t \to^{*}_{\mathtt{dB}} u$ then there exists r such that $s \to^{*}_{\mathtt{dB}} r \,{}^{*}_{\mathtt{ES}}\!\leftarrow u$.
- Again, commutation is a global property that can be obtained via a local one. In this case there are no lemmas analogous to Newman (commutation is more general than confluence), but the fact that $\to_{\mathtt{dB}}$ cannot duplicate $\to_{\mathtt{ES}}$ implies that their local commutation diagram trivially lifts to global commutation.
- Then $\to_{\mathtt{dB}} \cup \to_{\mathtt{ES}}$ is confluent.

This proof scheme was used for the first time by Accattoli and Paolini in [24]. In [3], Accattoli uses the local termination principle to prove the head factorisation theorem for the LSC in a way that is impossible in the λ-calculus.

The moral is that introducing sharing enables rewriting proof techniques that are impossible without it.

4.3 Linear Substitutions, Weak Evaluation, and Garbage Collection

Let's now decompose the substitution rule and define the LSC. Most of the literature of ES decomposes the substitution process using rules that are entangled with commuting rules such as $\to_{\lambda}$ and $\to_{@}$ described above. The special ingredient of the LSC is that substitution is decomposed but also disentangled from the commuting process. To properly define the rules we need to introduce a general notion of context:

$$\text{ES CONTEXTS} \qquad C, D \quad ::= \quad \langle\cdot\rangle \mid \lambda x.C \mid Ct \mid tC \mid C[x{\leftarrow}t] \mid t[x{\leftarrow}C]$$

Now, there are only two rules for evaluating explicit substitutions at a distance (plus $\mathtt{dB}$, to create them):

$$\begin{array}{lrcll} \text{LINEAR SUBSTITUTION} & C\langle\!\langle x\rangle\!\rangle[x{\leftarrow}s] & \to_{\mathtt{ls}} & C\langle\!\langle t\rangle\!\rangle[x{\leftarrow}s] & \\ \text{GARBAGE COLLECTION} & t[x{\leftarrow}s] & \to_{\mathtt{gc}} & t & \text{if } x \notin \mathtt{fv}(t) \end{array}$$

The idea is that given $t[x{\leftarrow}s]$ either x has no occurrences in t, and so the garbage collection rule $\to_{\mathtt{gc}}$ applies, or x does occur in t, which can then be written as $C\langle x\rangle$ for some context C (not capturing x), potentially in more than one way. The linear substitution rule then allows to replace the occurrence of x isolated by C without moving the ES $[x{\leftarrow}s]$.

The LSC is given by rules $\to_{\mathtt{dB}}$, $\to_{\mathtt{ls}}$, and $\to_{\mathtt{gc}}$. More precisely, the definition of these rules includes a further contextual closure (as it is standard also in the λ-calculus), that is, one defines $C\langle t\rangle \to_{\mathtt{dB}} C\langle s\rangle$ if $t \to_{\mathtt{dB}} s$, and similarly for the other rules.

Confluence for the LSC can be proved following exactly the same schema used for the substitution calculus.

Confluence of Weak Evaluation. One of the nice features of the LSC is that its definition is parametric in the grammar of contexts. For instance, we can define the Weak LSC by simply restricting general contexts C (used both to define rule $\to_{\mathtt{ls}}$ at top level and to give the contextual closure of the three rules) to weak contexts W that do not enter into abstractions:

$$\text{Weak ES contexts} \qquad W, W' \quad ::= \quad \langle\cdot\rangle \mid Wt \mid tW \mid W[x{\leftarrow}t] \mid t[x{\leftarrow}W]$$

The Weak LSC is confluent, and the proof goes always along the same lines.

Note that this is in striking contrast to what happens in the λ-calculus. Defining the weak λ-calculus by simply restricting the contextual closure to weak contexts produces a non-confluent calculus. For instance, the following diagram cannot be closed:

$$(\lambda x.\lambda y.yx)I \;{}_{\beta}{\leftarrow}\; (\lambda x.\lambda y.yx)(II) \to_{\beta} \lambda y.(y(II))$$

because II in the right reduct occurs under abstraction and it is then frozen. There are solutions to this issue, see Lévy and Maranget's [68], but they are all ad-hoc. In the LSC, instead, everything is as natural as it can possibly be.

Garbage Collection. Another natural property that holds in the LSC but not in the λ-calculus is the postponement of garbage collection. Rule $\to_{\mathtt{gc}}$ can indeed be postponed, that is, one has that if $t \to^*_{LSC} s$ then $t \to^*_{\mathtt{dB},\mathtt{ls}} \to^*_{\mathtt{gc}} s$. Since $\to_{\mathtt{gc}}$ is also strongly normalising, it is then *safe* to remove it and only consider the two other rules, $\to_{\mathtt{dB}}$ and $\to_{\mathtt{ls}}$.

The postponement of garbage collection models the fact that in programming languages the garbage collector acts asynchronously with respect to the execution flow.

In the λ-calculus, *erasing steps* are the analogous of garbage collection. A step $(\lambda x.t)s \to_{\beta} t\{x{\leftarrow}s\}$ is erasing if – like for garbage collection – the abstracted variable x does not occur in t, so that $t\{x{\leftarrow}s\} = t$, and s is simply erased.

The key point is that erasing steps cannot be postponed in the λ-calculus. Consider indeed the following sequence:

$$(\lambda x.\lambda y.y)ts \to_{\beta} (\lambda y.y)s \to_{\beta} s$$

The first step is erasing but it *cannot* be postponed after the second one, because the second step is *created* by the first one.

A Bird's Eye View. The list of unique elegant properties of the LSC is long. For instance, it has deep and simple connections to linear logic proof nets [8], the π-calculus [4], abstract machines and CbNeed [9], reasonable cost models [16, 18], open CbV [20], linear head reduction [3], multi types [19], it admits a residual system and a rich theory of standardisation [12], and even Lévy optimality [30]. Essentially, it is the canonical decomposition of the λ-calculus, and, in many respects, it is more expressive and flexible than the λ-calculus – it is a sort of λ-calculus 2.0.

Should then the LSC replace the λ-calculus? No. The point is not which system is the best. The point is acknowledging that the field is rich, and that even the simplest higher-order framework declines itself in a multitude of ways (weak/open/strong, cbn/cbv/cbneed, no sharing/one shot sharing/linear sharing), each one with its features and being a piece of a great puzzle.

References

1 Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit Substitutions. *J. Funct. Program.*, 1(4):375–416, 1991. doi:10.1017/S0956796800000186.

2 Samson Abramsky and C.-H. Luke Ong. Full Abstraction in the Lazy Lambda Calculus. *Inf. Comput.*, 105(2):159–267, 1993.

3 Beniamino Accattoli. An Abstract Factorization Theorem for Explicit Substitutions. In *RTA*, pages 6–21, 2012. doi:10.4230/LIPIcs.RTA.2012.6.

4 Beniamino Accattoli. Evaluating functions as processes. In *TERMGRAPH*, pages 41–55, 2013. doi:10.4204/EPTCS.110.6.

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

6 Beniamino Accattoli. The Complexity of Abstract Machines. In *WPTE@FSCD 2016*, pages 1–15, 2016. doi:10.4204/EPTCS.235.1.

7 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.

8 Beniamino Accattoli. Proof Nets and the Linear Substitution Calculus. In *ICTAC 2018*, pages 37–61, 2018. doi:10.1007/978-3-030-02508-3_3.

9 Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. Distilling abstract machines. In *ICFP 2014*, pages 363–376, 2014. doi:10.1145/2628136.2628154.

10 Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. A Strong Distillery. In *APLAS 2015*, pages 231–250, 2015. doi:10.1007/978-3-319-26529-2_13.

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

12 Beniamino Accattoli, Eduardo Bonelli, Delia Kesner, and Carlos Lombardi. 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*, pages 659–670, 2014. doi:10.1145/2535838.2535886.

13 Beniamino Accattoli and Claudio Sacerdoti Coen. On the Relative Usefulness of Fireballs. In *LICS 2015*, pages 141–155, 2015. doi:10.1109/LICS.2015.23.

14 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.

15 Beniamino Accattoli, Andrea Condoluci, Giulio Guerrieri, and Claudio Sacerdoti Coen. Crumbling Abstract Machines. Submitted, 2019.

16 Beniamino Accattoli and Ugo Dal Lago. On the Invariance of the Unitary Cost Model for Head Reduction. In *RTA*, pages 22–37, 2012. doi:10.4230/LIPIcs.RTA.2012.22.

17 Beniamino Accattoli and Ugo Dal Lago. Beta reduction is invariant, indeed. In *CSL-LICS '14*, pages 8:1–8:10, 2014. doi:10.1145/2603088.2603105.

18 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.

19 Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds. *PACMPL*, 2(ICFP):94:1–94:30, 2018. doi:10.1145/3236789.

20 Beniamino Accattoli and Giulio Guerrieri. Open Call-by-Value. In *APLAS 2016*, pages 206–226, 2016. doi:10.1007/978-3-319-47958-3_12.

21 Beniamino Accattoli and Giulio Guerrieri. Implementing Open Call-by-Value. In *FSEN 2017, Tehran, Iran, April 26-28, 2017, Revised Selected Papers*, pages 1–19, 2017. doi:10.1007/978-3-319-68972-2_1.

22 Beniamino Accattoli and Giulio Guerrieri. Types of Fireballs. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, pages 45–66, 2018. doi:10.1007/978-3-030-02768-1_3.

23 Beniamino Accattoli, Giulio Guerrieri, and Maico Leberle. 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, pages 410–439, 2019. doi:10.1007/978-3-030-17184-1_15.

24 Beniamino Accattoli and Luca Paolini. Call-by-Value Solvability, revisited. In *FLOPS*, pages 4–16, 2012. doi:10.1007/978-3-642-29822-6_4.

25 Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. The Call-by-Need Lambda Calculus. In *POPL'95*, pages 233–246, 1995. doi:10.1145/199448.199507.

26 Federico Aschieri. Game Semantics and the Geometry of Backtracking: a New Complexity Analysis of Interaction. *J. Symb. Log.*, 82(2):672–708, 2017. doi:10.1017/jsl.2016.48.

27 Andrea Asperti and Harry G. Mairson. Parallel Beta Reduction is not Elementary Recursive. In *POPL*, pages 303–315, 1998. doi:10.1145/268946.268971.

28 Patrick Baillot. Stratified coherence spaces: a denotational semantics for light linear logic. *Theor. Comput. Sci.*, 318(1-2):29–55, 2004. doi:10.1016/j.tcs.2003.10.015.

29 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.

30 Pablo Barenbaum and Eduardo Bonelli. Optimality and the Linear Substitution Calculus. In *FSCD 2017*, pages 9:1–9:16, 2017. doi:10.4230/LIPIcs.FSCD.2017.9.

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

32 Guy E. Blelloch and John Greiner. Parallelism in Sequential Functional Languages. In *FPCA*, pages 226–237, 1995. doi:10.1145/224164.224210.

33 Alberto Carraro and Giulio Guerrieri. 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*, pages 103–118, 2014. doi:10.1007/978-3-642-54830-7_7.

34 Pierre Clairambault. Estimation of the Length of Interactions in Arena Game Semantics. In *FOSSACS*, pages 335–349, 2011. doi:10.1007/978-3-642-19805-2_23.

35 Pierre Clairambault. Bounding Skeletons, Locally Scoped Terms and Exact Bounds for Linear Head Reduction. In *TLCA*, pages 109–124, 2013. doi:10.1007/978-3-642-38946-7_10.

36 Pierre Crégut. An Abstract Machine for Lambda-Terms Normalization. In *LISP and Functional Programming*, pages 333–340, 1990.

37 Pierre Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3):209–230, 2007. doi:10.1007/s10990-007-9015-z.

38 Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ICFP*, pages 233–243, 2000. doi:10.1145/351240.351262.

39 Ugo Dal Lago and Simone Martini. An Invariant Cost Model for the Lambda Calculus. In *CiE 2006*, pages 105–114, 2006. doi:10.1007/11780342_11.

40 Ugo Dal Lago and Simone Martini. Derivational Complexity Is an Invariant Cost Model. In *FOPARA 2009*, pages 100–113, 2009. doi:10.1007/978-3-642-15331-0_7.

41 Ugo Dal Lago and Simone Martini. On Constructor Rewrite Systems and the Lambda-Calculus. In *ICALP (2)*, pages 163–174, 2009. doi:10.1007/978-3-642-02930-1_14.

42 Vincent Danos, Hugo Herbelin, and Laurent Regnier. Game Semantics & Abstract Machines. In *LICS*, pages 394–405, 1996. doi:10.1109/LICS.1996.561456.

43 Vincent Danos and Laurent Regnier. Reversible, Irreversible and Optimal lambda-Machines. *Theor. Comput. Sci.*, 227(1-2):79–97, 1999. doi:10.1016/S0304-3975(99)00049-3.

44 Daniel de Carvalho. Execution time of λ-terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science*, 28(7):1169–1203, 2018. doi:10.1017/S0960129516000396.

45 Daniel de Carvalho, Michele Pagani, and Lorenzo Tortora de Falco. A semantic measure of the execution time in linear logic. *Theor. Comput. Sci.*, 412(20):1884–1902, 2011. doi:10.1016/j.tcs.2010.12.017.

46 Roy Dyckhoff and Stéphane Lengrand. Call-by-Value lambda-calculus and LJQ. *J. Log. Comput.*, 17(6):1109–1134, 2007. doi:10.1093/logcom/exm037.

47 Thomas Ehrhard. Collapsing non-idempotent intersection types. In *CSL*, pages 259–273, 2012. doi:10.4230/LIPIcs.CSL.2012.259.

48 Dan R. Ghica. Slot games: a quantitative model of computation. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 85–97, 2005. doi:10.1145/1040305.1040313.

49 Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

50 Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *(ICFP '02)*., pages 235–246, 2002. doi:10.1145/581478.581501.

51 Hugo Herbelin and Stéphane Zimmermann. An Operational Account of Call-by-Value Minimal and Classical lambda-Calculus in "Natural Deduction" Form. In *TLCA*, pages 142–156, 2009. doi:10.1007/978-3-642-02273-9_12.

52 Andrew Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 177–190, 2007. doi:10.1145/1291151.1291179.

53 Delia Kesner. Reasoning About Call-by-need by Means of Types. In *FOSSACS 2016*, pages 424–441, 2016. doi:10.1007/978-3-662-49630-5_25.

54 Jean-Louis Krivine. *Lambda-calculus, types and models*. Ellis Horwood series in computers and their applications. Masson, 1993.

55 Arne Kutzner and Manfred Schmidt-Schauß. A Non-Deterministic Call-by-Need Lambda Calculus. In *ICFP 1998*, pages 324–335, 1998. doi:10.1145/289423.289462.

56 Ugo Dal Lago, Claudia Faggian, Ichiro Hasuo, and Akira Yoshimizu. The geometry of synchronization. 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*, pages 35:1–35:10, 2014. doi:10.1145/2603088.2603154.

57 Ugo Dal Lago, Claudia Faggian, Benoît Valiron, and Akira Yoshimizu. Parallelism and Synchronization in an Infinitary Context. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 559–572, 2015. doi:10.1109/LICS.2015.58.

58 Ugo Dal Lago, Claudia Faggian, Benoît Valiron, and Akira Yoshimizu. The geometry of parallelism: classical, probabilistic, and quantum effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 833–845, 2017. URL: http://dl.acm.org/citation.cfm?id=3009859.

59 Ugo Dal Lago and Martin Hofmann. Quantitative Models and Implicit Complexity. In *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings*, pages 189–200, 2005. doi:10.1007/11590156_15.

60 Ugo Dal Lago and Olivier Laurent. Quantitative Game Semantics for Linear Logic. In *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings*, pages 230–245, 2008. doi:10.1007/978-3-540-87531-4_18.

61 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.

62 Ugo Dal Lago, Ryo Tanaka, and Akira Yoshimizu. The geometry of concurrent interaction: Handling multiple ports by way of multiple tokens. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005112.

63 Jim Laird, Giulio Manzonetto, Guy McCusker, and Michele Pagani. Weighted Relational Models of Typed Lambda-Calculi. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 301–310, 2013. doi:10.1109/LICS.2013.36.

64 John Lamping. An Algorithm for Optimal Lambda Calculus Reduction. In *POPL*, pages 16–30, 1990. doi:10.1145/96709.96711.

65 John Launchbury. A Natural Semantics for Lazy Evaluation. In *POPL*, pages 144–154, 1993. doi:10.1145/158511.158618.

66 Olivier Laurent and Lorenzo Tortora de Falco. Obsessional Cliques: A Semantic Characterization of Bounded Time Complexity. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 179–188, 2006. doi:10.1109/LICS.2006.37.

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

68 Jean-Jacques Lévy and Luc Maranget. Explicit Substitutions and Programming Languages. In *Foundations of Software Technology and Theoretical Computer Science, 19th Conference, Chennai, India, December 13-15, 1999, Proceedings*, pages 181–200, 1999. doi:10.1007/3-540-46691-6_14.

69 Ian Mackie. The Geometry of Interaction Machine. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 198–208, 1995. doi:10.1145/199448.199483.

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

71 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.

72 Damiano Mazza. Simple Parsimonious Types and Logarithmic Space. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*, volume 41 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24–40, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CSL.2015.24.

73 Robin Milner. Functions as Processes. *Math. Str. in Comput. Sci.*, 2(2):119–141, 1992. doi:10.1017/S0960129500001407.

74 Robin Milner. Local Bigraphs and Confluence: Two Conjectures. *Electr. Notes Theor. Comput. Sci.*, 175(3):65–73, 2007. doi:10.1016/j.entcs.2006.07.035.

75 Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I. *Inf. Comput.*, 100(1):1–40, 1992. doi:10.1016/0890-5401(92)90008-4.

76 Eugenio Moggi. Computational λ-Calculus and Monads. In *LICS '89*, pages 14–23, 1989.

77 Andrzej S. Murawski and C.-H. Luke Ong. Discreet Games, Light Affine Logic and PTIME Computation. In *Computer Science Logic, 14th Annual Conference of the EACSL, Fischbachau, Germany, August 21-26, 2000, Proceedings*, pages 427–441, 2000. doi:10.1007/3-540-44622-2_29.

78 Koko Muroya and Dan R. Ghica. The Dynamic Geometry of Interaction Machine: A Call-by-Need Graph Rewriter. In *CSL 2017*, pages 32:1–32:15, 2017. doi:10.4230/LIPIcs.CSL.2017.32.

79 Luca Paolini. Call-by-Value Separability and Computability. In *ICTCS*, pages 74–89, 2001. doi:10.1007/3-540-45446-2_5.

80 Luca Paolini and Simona Ronchi Della Rocca. Call-by-value Solvability. *ITA*, 33(6):507–534, 1999. doi:10.1051/ita:1999130.

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

82 Simona Ronchi Della Rocca and Luca Paolini. *The Parametric λ-Calculus*. Springer Berlin Heidelberg, 2004.

83 Amr Sabry and Matthias Felleisen. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993. doi:10.1007/BF01019462.

84 Amr Sabry and Philip Wadler. A Reflection on Call-by-Value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941, 1997. doi:10.1145/267959.269968.

85 David Sands, Jörgen Gustavsson, and Andrew Moran. Lambda Calculi and Linear Speedups. In *The Essence of Computation*, pages 60–84, 2002. doi:10.1007/3-540-36377-7_4.

86 Ulrich Schöpp. Space-Efficient Computation by Interaction. In *CSL 2006*, pages 606–621, 2006. doi:10.1007/11874683_40.

87 Peter Sestoft. Deriving a Lazy Abstract Machine. *J. Funct. Program.*, 7(3):231–264, 1997. URL: http://journals.cambridge.org/action/displayAbstract?aid=44087.

88 Cees F. Slot and Peter van Emde Boas. On Tape Versus Core; An Application of Space Efficient Perfect Hash Functions to the Invariance of Space. In *STOC 1984*, pages 391–400, 1984. doi:10.1145/800057.808705.

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