The Negligible and Yet Subtle Cost of Pattern Matching

Beniamino Accattoli, Bruno Barras

The Negligible and Yet Subtle Cost of Pattern Matching

Beniamino Accattoli and Bruno Barras

INRIA, UMR 7161, LIX, École Polytechnique,
beniamino.accattoli@inria.fr
bruno.barras@inria.fr

Abstract. The model behind functional programming languages is the *closed λ-calculus*, that is, the fragment of the λ-calculus where evaluation is weak (*i.e.* out of abstractions) and terms are closed. It is well-known that the number of β (*i.e.* evaluation) steps is a reasonable cost model in this setting, for all natural evaluation strategies (call-by-name / value / need). In this paper we try to close the gap between the closed λ-calculus and actual languages, by considering an extension of the λ-calculus with pattern matching. It is straightforward to prove that β plus matching steps provide a reasonable cost model. What we do then is finer: we show that β steps only, without matching steps, provide a reasonable cost model also in this extended setting—morally, pattern matching comes for free, complexity-wise. The result is proven for all evaluation strategies (name / value / need), and, while the proof itself is simple, the problem is shown to be subtle. In particular we show that qualitatively equivalent definitions of call-by-need may behave very differently.

This work is part of a wider research effort, the COCA HOLA project [3].

1 Introduction

Functional programming languages are modeled on the λ-calculus. More precisely, on the dialect in which evaluation is *weak*, that is, it does not enter function bodies, and terms are closed—we refer to this setting as to the *closed λ-calculus*. In contrast to other models such as Turing machines, in the λ-calculus it is far from evident that the number of evaluation steps is a reasonable cost model for time. Its evaluation rule, β-reduction, is in fact a complex, non-atomic operation, for which there exist *size exploding families*, *i.e.* families of programs whose code grows at an exponential rate with respect to the number of β-reductions.

The time cost models of the closed λ-calculus. Since the work of Blelloch and Greiner [17], it is known that the number of β-steps in the call-by-value closed λ-calculus can indeed be considered as a reasonable cost model. Roughly, one can consider β as an (almost) atomic operation, counting 1 (actually a cost bound by the size of the initial term) for each step. The key point is that β can be

simulated efficiently, using simple forms of shared evaluation such as environment-based abstract machines, circumventing size explosion. Sands, Gustavsson, and Moran have then showed that ordinary abstract machines for call-by-name and call-by-need closed λ-calculi are also reasonable [34]. Similar results have also been obtained by Martini and Dal Lago (by combining the results in [22] and [21]), and then the whole question has been finely decomposed and studied by Accattoli, Barenbaum, Mazza, and Sacerdoti Coen [6,14]. It is thus fair to say that the number of β-steps is *the* time cost model of the closed λ-calculus.

Functional programming languages and the negligible cost hypothesis. There is a gap between the closed λ-calculus and an actual functional language, that usually has various constructs and evaluation rules in addition to β-reduction. From the cited results it would easily follow that the number of β-steps plus the steps for the additional constructs is a reasonable cost model. In practice, however, a more parsimonious cost model is used: for functional programs the number of function calls (aka β-reductions) is usually considered as (an upper bound on) its time complexity—this is done for instance by Charguéraud and Pottier in [18]. The implicit hypothesis is that the cost of β-reduction dominates the cost of all these additional rules, so that it is fair to ignore them, complexity-wise—that is, they can be considered to have zero cost. To the best of our knowledge, however, such a *negligible cost hypothesis* has never been proved.

The cost of pattern matching. This paper is a first step towards proving the negligible cost hypothesis. Here, we extend the study of cost models to the closed λ-calculus with constructors and pattern matching. It turns out that the problem is subtler than what folklore suggests: evaluation steps related to pattern matching can easily be exponential in the number of β-steps—*i.e.* they are far from being dominated by β. We show, however, that evaluation can be *simulated* so that matching steps are tamed. The cost of pattern matching is proved to be indeed negligible: matching steps can be assigned zero cost, as they are linear in the number of β-steps and the size of the initial term, the two key parameters in the study of cost models. Therefore, our results provide formal arguments supporting common practice, despite the apparently bad behavior of pattern matching.

In contrast to the ordinary closed λ-calculus, where call-by-name / value / need strategies can be treated with the same techniques, for pattern matching these evaluation strategies behave less uniformly. Namely:

- *Call-by-name (CbN) explodes*: we show that in CbN there are *matching exploding families*, that is, families $\{t_n\}_{n\in\mathbb{N}}$ of terms where t_n evaluates in n β-steps and 2^n matching steps, suggesting that the cost of pattern matching is far from being negligible.
- *Call-by-value (CbV) is reasonable*: the explosion of matching steps in CbN is connected to the re-evaluation of function arguments, it is then natural to look at the CbV case, where arguments are evaluated once and for all. It turns out that in CbV matchings are negligible, namely they are *bilinear*,

that is, linear in the number of β-steps and the size of the initial term, and that such a bound is tight.
- *Call-by-need (CbNeed) is sometimes reasonable*: CbNeed is halfway CbN and CbV, as it is operationally equivalent to CbN but it avoids the re-evaluation of arguments as in CbV—in particular CbNeed rests on *values*. The problem here is subtle, and amounts to how values are defined. If constructors are always considered as values, independently of the shape of their arguments, then there are matching exploding families similar—but trickier—to those affecting CbN. If constructors are considered as values only when they are applied to variables, then one can adapt the proof used for CbV, and show that matchings are negligible.
- *Call-by-name (CbN) is reasonable, actually*: being operationally equivalent to CbN, CbNeed can be seen as an efficient simulation of CbN, proving that the matching exploding families of CbN are a circumventable problem, similarly to the size exploding families for β-reduction.

The context of the paper. The problem of the cost of pattern matching arises as an intermediate steps in a more ambitious research program, going beyond the negligible cost hypothesis. Our real goal in fact is the complexity analysis of the abstract machine at work in the kernel of Coq[1] [20]. Such a machine has been designed and partially studied by Barras in his PhD thesis [16], and provides a lightweight approach compared to the compilation scheme by Grégoire and Leroy described in [25]. It is used to decide convertibility of terms, which is the bottleneck of the type-checking (and thus proof-checking) algorithm. It is at the same time one of the most sophisticated and one of the most used abstract machines for the λ-calculus. The goal is to prove it *reasonable*, that is, to show that the overhead of the machine is polynomial in the number of β-steps and in the size of the initial term, and eventually design a new machine along the way, if the existing one turns out to be unreasonable.

Barras' machine executes a language that is richer than λ-calculus. In particular, it includes constructors and pattern matching, to which the paper is devoted—this justifies the choice of the particular presentation of pattern matching that we adopt, rather than other formalisms such as Cirstea and Kirchner's *rewriting calculus* [19], Klop, van Oostrom, and de Vrijer's *λ-calculus with patterns* [29], or Jay and Kesner's *pure pattern calculus* [26]. The machine actually implements call-by-need strong (*i.e.* under abstraction) evaluation, while here we only deal with the closed case. This is done for the sake of simplicity, because the subtleties concerning pattern matching are already visible at the closed level, but also because the closed case is of wider interest, being the one modeling functional programming languages.

The value of the paper. To our knowledge, our work is the first study of the asymptotic cost of pattern matching in a functional setting. As we explained,

[1] The kernel of Coq is the subset of the codebase which ensures that only valid proofs are accepted. Hence the use of an abstract machine, which has a better ratio efficiency/complexity than the use of a compiler or a naive interpreter.

this paper provides an example of the subtleties hidden in passing from the ideal, abstract setting of the closed λ-calculus to an actual, concrete functional language—and the case we study here is still quite abstract—motivating further complexity analyses of programming features beyond the core of the λ-calculus. Another interesting point is the fact that the study of cost models is used to discriminate between different presentations of CbNeed that would otherwise seem equivalent. Said differently, complexity and cost models are used here as language design principles.

The style of the paper. We adopt a lightweight, minimal style, focusing on communicating ideas rather than providing a comprehensive treatment of the calculi under study. The style is akin to that of a *functional pearl*—the reasoning is in fact simple, not far from a pearl. A more thorough study is left to an eventual longer version of this work. In particular, our results are proved using simple calculi with explicit substitutions (ES) inspired by the *linear substitution calculus*—a variation over a λ-calculus with ES by Robin Milner [32] developed by Accattoli and Kesner [2,8]—in which both the search of the redex and α-renaming are left to the meta-level. To be formal, we should make both tasks explicit in the form of an abstract machine. The work of Accattoli and coauthors [6,9,4] has however repeatedly showed that these tasks require an overhead linear in the number of β-steps and the size of the initial term, and in some cases even logarithmic in the size of the initial term (see the companion paper [7])—in the terminology of this paper, the costs of *search* and *α-renaming* are negligible.

At the technical level, for the study of cost models we mostly adopt the techniques and the terminology (linear substitution calculus, subterm invariant, harmony, etc) developed by Accattoli and his coauthors (Dal Lago, Barenbaum, Mazza, Sacerdoti Coen, Guerrieri) in [10,6,9,11,4].

2 Call-by-Name and Matching Explosion

Here we consider the case of the CbN closed λ-calculus extended with constructors and pattern matching. Since the aim is to show a degeneracy, we proceed quickly (omitting the error handling, for instance), delaying a more formal treatment to the next section on CbV, where we show a positive result.

Constructors and pattern matching. The language is the ordinary λ-calculus extended with a fixed finite set of constructors $\mathtt{c}_1, \dots \mathtt{c}_k$—therefore, k is a constant parameter of the language. Each constructor $\mathtt{c}_i$ takes a fixed number $k_{\mathtt{c}_i}$ (≥ 0) of arguments, *e.g.* $\mathtt{c}_i(t_1, \dots, t_{k_{\mathtt{c}_i}})$. Constructors are supposed to be fully applied from the beginning, and the application of constructors to their arguments is *not* the application of the λ-calculus—that is, we write $\mathtt{c}_i(t_1, \dots, t_{k_{\mathtt{c}_i}})$ and not $\mathtt{c}_i t_1, \dots, t_{k_{\mathtt{c}_i}}$, thus ruling out partial applications such as *e.g.* $\mathtt{c}_i t_1$ (assuming that $k_{\mathtt{c}_i} > 1$). There also is a pattern matching operator $\mathtt{case}\, t\, \{b\}$, where b is a set of branches—since k is fixed, for the sake of simplicity we assume that every

$\mathtt{case}\, t\, \{b\}$ has a branch for every constructor. Namely:

$$\begin{array}{rr} \textsc{Terms} & t, u ::= x \mid \lambda x.t \mid t\ u \mid \mathtt{c}_i(\boldsymbol{t}) \mid \mathtt{case}\, t\, \{b\} \\ \textsc{Branches} & b ::= \mathtt{c}_{\boldsymbol{i}}(\boldsymbol{x}) \Rightarrow \boldsymbol{u_i} \end{array}$$

where the bold font denotes vectors according to the following conventions:

- $\boldsymbol{t}$ is the notation for a vector of terms $t_1, \ldots, t_n$, whose length is left implicit as much as possible;
- $\mathtt{c}_i(\boldsymbol{t})$ and $\mathtt{c}_i(\boldsymbol{x})$ assume that $\boldsymbol{t}$ and $\boldsymbol{x}$ have the right arity $k_{\mathtt{c}_i}$, and
- $\mathtt{c}_{\boldsymbol{i}}(\boldsymbol{x}) \Rightarrow \boldsymbol{u_i}$ is a compact notation for $\mathtt{c}_1(\boldsymbol{x}) \Rightarrow u_1, \ldots, \mathtt{c}_k(\boldsymbol{x}) \Rightarrow u_k$.

Moreover, $\mathtt{c}_i(x_1, \ldots, x_{k_{\mathtt{c}_i}}) \Rightarrow u_i$ binds the variables $x_1, \ldots, x_{k_{\mathtt{c}_i}}$ in u_i. Finally, the size of a term is the number of its term constructors (that is, the number of productions used to derive it using the grammar for terms), and it is noted $|t|$.

Evaluation. The small-steps operational semantics is the usual one for CbN, extended with an evaluation context and a rewriting rule for pattern matching.

$$\textsc{CbN Evaluation Contexts} \quad C ::= \langle\cdot\rangle \mid Ct \mid \mathtt{case}\, C\, \{b\}$$

$$\begin{array}{cc} \textsc{Rule at Top Level} & \textsc{Contextual closure} \\ (\lambda x.t)u \mapsto_\beta t\{x \leftarrow u\} & C\langle t\rangle \to_\beta C\langle u\rangle \quad \text{if } t \mapsto_\beta u \\ \mathtt{case}\, \mathtt{c}_i(\boldsymbol{t})\, \{\mathtt{c}_{\boldsymbol{i}}(\boldsymbol{x}) \Rightarrow \boldsymbol{u_i}\} \mapsto_{\mathtt{case}} u_i\{\boldsymbol{x} \leftarrow \boldsymbol{t}\} & C\langle t\rangle \to_{\mathtt{case}} C\langle u\rangle \quad \text{if } t \mapsto_{\mathtt{case}} u \end{array}$$

To help the reader getting used to our notations, let us unfold the $\mapsto_{\mathtt{case}}$ rule:

$$\begin{array}{c} \mathtt{case}\, \mathtt{c}_i(t_1, \ldots, t_{k_{\mathtt{c}_i}})\, \{\mathtt{c}_1(\boldsymbol{x}) \Rightarrow u_1, \ldots, \mathtt{c}_k(\boldsymbol{x}) \Rightarrow u_k\} \mapsto_{\mathtt{case}} \\ u_i\{x_1 \leftarrow t_1\} \ldots \{x_{k_{\mathtt{c}_i}} \leftarrow t_{k_{\mathtt{c}_i}}\} \end{array}$$

The union of $\to_\beta$ and $\to_{\mathtt{case}}$ is noted $\to_{\mathtt{CbN}}$. A derivation $d : t \to^* u$ is a potentially empty sequence of evaluation $\to_\beta$ and $\to_{\mathtt{case}}$ steps, whose length / number of β steps / number of $\to_{\mathtt{case}}$ steps is denoted by $|d|$ / $|d|_\beta$ / $|d|_{\mathtt{case}}$. As it is standard, we silently work modulo α-equivalence.

The matching exploding family. We already have enough ingredients to build a matching exploding family. Consider a a zeroary constructor 0. Now, define the following family of closed terms:

$$\begin{array}{c} t_1 := \lambda x.\mathtt{case}\, x\, \{0 \Rightarrow \mathtt{case}\, x\, \{0 \Rightarrow 0\}\} \\ t_{n+1} := \lambda x.(t_n(\mathtt{case}\, x\, \{0 \Rightarrow \mathtt{case}\, x\, \{0 \Rightarrow 0\}\})) \end{array}$$

Our exploding family is actually given by $\{t_n\, 0\}_{n\in\mathbb{N}}$, for which we want to prove that $t_n 0 \to_\beta^n \to_{\mathtt{case}}^{2^n} 0$. To this aim, we need the following auxiliary family:

$$u_0 := 0 \qquad u_{n+1} := \mathtt{case}\, u_n\, \{0 \Rightarrow \mathtt{case}\, u_n\, \{0 \Rightarrow 0\}\})$$

Now, in two steps, we prove a slightly more general statement, namely $t_n u_k \to_\beta^n u_{n+k} \to_{\mathtt{case}}^{2^{n+k}} 0$.

Proposition 1.

1. Linear β prefix*: there exists a derivation* $d_n : t_n u_k \to_\beta^n u_{n+k}$ *for* $n \geq 1$ *and* $k \geq 0$*;*
2. Exponential pattern matching suffix*: there exists a derivation* $e_n : u_n \to_{\mathsf{case}}^* 0$ *with* $|e_n| = \Omega(2^n)$ *for* $n \geq 1$.
3. Matching exploding family*: there exists a derivation* $f_n : t_n 0 \to^* 0$ *with* $|t_n 0| = O(n)$*,* $|f_n|_\beta = n$*, and* $|f_n| = \Omega(2^n)$.

Proof. Point 3 is obtained by concatenating Point 1 and Point 2.
Point 1 and Point 2 are by induction on n. Cases:

- *Base case, i.e.* $n = 1$.
  1. *Linear* β *prefix*: the derivation d_1 is given by

$$\begin{array}{rcll} t_1 u_k & = & (\lambda x.\mathsf{case}\, x\, \{0 \Rightarrow \mathsf{case}\, x\, \{0 \Rightarrow 0\}\}) u_k & \\ & \to_\beta & \mathsf{case}\, u_k\, \{0 \Rightarrow \mathsf{case}\, u_k\, \{0 \Rightarrow 0\}\} & = u_{k+1} \end{array}$$

  2. *Exponential pattern matching suffix*: the derivation e_1 given by the following sequence has indeed $2^1 = 2$ steps, as required:

$$\begin{array}{rcll} u_1 & = & \mathsf{case}\, 0\, \{0 \Rightarrow \mathsf{case}\, 0\, \{0 \Rightarrow 0\}\}) & \\ & \to_{\mathsf{case}} & \mathsf{case}\, 0\, \{0 \Rightarrow 0\} & \to_{\mathsf{case}} 0 \end{array}$$

- *Inductive case.*
  1. *Linear* β *prefix*: the derivation d_{n+1} is given by

$$\begin{array}{rcl} t_{n+1} u_k & = & (\lambda x.(t_n \mathsf{case}\, x\, \{0 \Rightarrow \mathsf{case}\, x\, \{0 \Rightarrow 0\}\})) u_k \\ & \to_\beta & t_n \mathsf{case}\, u_k\, \{0 \Rightarrow \mathsf{case}\, u_k\, \{0 \Rightarrow 0\}\})) \\ & = & t_n u_{k+1} \\ (\text{by } \textit{i.h.}) & \overset{d_n}{\to_\beta^n} & u_{n+k+1} \end{array}$$

  2. *Exponential pattern matching suffix*: the derivation e_{n+1} is given by:

$$\begin{array}{rcll} u_{n+1} & = & \mathsf{case}\, u_n\, \{0 \Rightarrow \mathsf{case}\, u_n\, \{0 \Rightarrow 0\}\}) & \\ (\text{by } \textit{i.h.}) & \overset{e_n}{\to^*} & \mathsf{case}\, 0\, \{0 \Rightarrow \mathsf{case}\, u_n\, \{0 \Rightarrow 0\}\}) & \\ & \to_{\mathsf{case}} & \mathsf{case}\, u_n\, \{0 \Rightarrow 0\} & \\ (\text{by } \textit{i.h.}) & \overset{e_n}{\to^*} & \mathsf{case}\, 0\, \{0 \Rightarrow 0\} & \to_{\mathsf{case}} 0 \end{array}$$

  Now, $|e_{n+1}| = 2 + 2 \cdot |e_n| =_{i.h.} 2 + 2 \cdot \Omega(2^n) = \Omega(2^{n+1})$. □

3 Call-by-Value, LIME, and the Bilinear Bound

It is easy to see that the matching exploding family of the previous section does not explode if evaluated according to the CbV strategy. Said differently, the problem seems to be about the re-evaluation of arguments.

We prove here that for the CbV closed λ-calculus extended with constructors and pattern matching the number of β-steps (alone, *i.e.* without matching steps) is a reasonable cost model. Despite the absence of matching explosion, to reach our goal we have to address the underlying size explosion problem that affects every λ-calculus with a small-step operational semantics, and so we have to adopt sharing and an abstract machine-like formalism. In the terminology of the introduction, we have to define a framework *simulating* the small-step calculus, in order to tame size explosion.

Therefore, we switch from a small-step operational semantics to a *micro*-step one, that is, we replace β-reduction $\to_\beta$ and meta-level substitution $t\{x \leftarrow u\}$ with a *multiplicative* rule $\to_{\mathtt{m}}$, turning a β-redex $(\lambda x.t)u$ into an explicit, delayed substitution $t[x \leftarrow u]$, and an *exponential* rule $\to_{\mathtt{e}}$, replacing one variable occurrence at the time, when it ends up in evaluation position. The terminology *multiplicative / exponential* comes from the connection with linear logic, that is however kept hidden here—see [6] for more details—just bear in mind that the exponential rule does not have an exponential cost, the name is due to other reasons.

For the sake of simplicity, we define the micro-step calculus but not the small-step one, and thus we also omit the study of the correspondence between the two. It would be obtained by simply unfolding the explicit substitutions (ES), and it is standard—see [4] for a detailed similar study in CbN. The only point that is important is that in such a correspondence there is a bijection between the evaluation steps at the two levels, except for the exponential steps, that vanish, because ES are unfolded by the correspondence. In particular, the number of multiplicative and β-steps coincide, and they can thus be identified for our complexity analyses.

Introducing LIME. Our proof uses a new simple formalism, the *LInear Matching calculus by valuE* (shortened *LIME*), that is a variation over other formalisms studied by Accattoli and coauthors (the *value substitution calculus* [13], the *GLAM abstract machine* [9], and the *micro-substituting abstract machine* [4]).

Let us explain how to classify LIME in the zoo of decompositions of the λ-calculus. There are three tasks that in the λ-calculus are left implicit or at the meta-level and that are addressed by finer frameworks such as abstract machines or calculi with ES:

1. *Substitution*: delaying and decomposing the substitution process;
2. *Search*: searching for the next redex to reduce;
3. *Names*: handling/avoiding α-renaming.

The original approach to calculi with ES [1] addressed all these tasks. With time, it was realized that the handling of names could be safely left implicit, see Kesner's [28] for a survey. More recently, also the search of the redex has been factored out, bringing it back to the implicit level, making ES act at a distance, without percolating through the term. The paradigmatic framework of this simpler, *at a distance* approach is the *linear substitution calculus* (LSC), a variation over a λ-calculus with ES by Robin Milner [32] developed by Accattoli and Kesner [2,8]—a LSC-like calculus is used in forthcoming Sect. 4. LIME, as

the LSC, addresses only the substitution task, letting the other two implicit. Here, however, we add a further simplification: it groups all ES in a global environment, in a way inspired by abstract machines and at work also in Accattoli's [4]. The literature of course contains also other formalisms employing a global environment or factoring out the search of the redex, at least [36,34,38,24,27,23,35], but usually developed focusing on other points.

The only data structure in LIME is the global environment E for delayed substitutions. With respect to abstract machines, the idea is that the transitions for the search of the redex are omitted (together with the related data structures, such as stacks and dumps), because the transitions corresponding to β, matching, and substitution transitions are expressed via evaluation contexts. For what concerns α-renaming, we follow the same approach used in the mainstream approach to the λ-calculus, leaving it at the meta-level and applying it on-the-fly. Our choice is justified by the fact that, as already pointed out in the introduction, previous work has repeatedly showed that the costs of handling *search* and *names* explicitly are negligible, when one is interested is showing that the overhead is not exponential.

Our result is that the number of steps of LIME is bilinear, that is, linear in the number of β-steps and the size of the initial term, that are the two fundamental parameters in the study of cost models. Additionally, we show that our bound is tight. Making search and names explicit usually has only an additional bilinear cost, that would not change the asymptotic behavior. The choice of omitting them, then, is particularly reasonable.

Defining LIME. The idea is that a term t is paired with an environment E, to form a program p. There is a special program $\mathtt{err}$, denoting that an error occurred, that can happen in two cases: because of a pattern matching on an abstraction, or the application of a constructor to a further argument—the two cases are spelled out by the forthcoming rewriting rules. Evaluation is right-to-left, and values include abstractions, error, and constructors applied recursively to values. In particular, variables are excluded from values as it is standard in the literature on abstract machines, see [14]. The language is thus defined by:

$$
\begin{array}{rrl}
\textsc{Terms} & t,u & ::= x \mid \lambda x.t \mid t\ u \mid \mathtt{c}(\boldsymbol{t}) \mid \mathtt{case}\, t\, \{b\} \mid \mathtt{err} \\
\textsc{Branches} & b & ::= \mathtt{c_i}(\boldsymbol{x}) \Rightarrow \boldsymbol{u_i} \\
\textsc{Values} & v,w & ::= \lambda x.t \mid \mathtt{c}(\boldsymbol{v}) \mid \mathtt{err} \\
\textsc{Environments} & E & ::= \epsilon \mid [x \leftarrow v] :: E \\
\textsc{Programs} & p & ::= (t, E) \\
\\
\textsc{Eval. Contexts} & C & ::= \langle\cdot\rangle \mid tC \mid Cv \mid \mathtt{c}(t,\dots,C,\dots,v) \mid \mathtt{case}\, C\, \{b\}
\end{array}
$$

Note that the definition of evaluation contexts forces the evaluation of constructor arguments, from right to left. Most of the time we write programs (t, E) without the parentheses, *i.e.* simply as $t\,E$. Evaluation $\rightarrow_{\mathtt{CbV}}$ is the relation obtained as the union of the following rewriting rules ($\mathtt{m}$ for *multiplicative* and $\mathtt{e}$ for *exponential*). They are not defined at top level and then closed by evaluation context but are defined directly at the global level (by means of evaluation contexts, of course):

$$\begin{array}{rll} C\langle(\lambda x.t)\ v\rangle\ E & \to_{\mathtt{m}} & C\langle t\rangle\ [x{\leftarrow}v] :: E \\ C\langle x\rangle\ E :: [x{\leftarrow}v] :: E' & \to_{\mathtt{e}} & C\langle v\rangle\ E :: [x{\leftarrow}v] :: E' \\ C\langle \mathtt{case}\ \mathtt{c}_i(\boldsymbol{v})\ \{\mathtt{c}_j(\boldsymbol{x}) \Rightarrow \boldsymbol{u_j}\}\rangle\ E & \to_{\mathtt{case}} & C\langle u_i\rangle\ [\boldsymbol{x}{\leftarrow}\boldsymbol{v}] :: E \\ C\langle \mathtt{case}\ \lambda x.t\ \{b\}\rangle\ E & \to_{\mathtt{err}_1} & \mathtt{err}\ \epsilon \\ C\langle \mathtt{c}_i(\boldsymbol{v})\ t\rangle\ E & \to_{\mathtt{err}_2} & \mathtt{err}\ \epsilon \end{array}$$

where rule $\to_{\mathtt{case}}$ has been written compactly. Its explicit form is

$$\begin{array}{l} C\langle \mathtt{case}\ \mathtt{c}_i(v_1, \ldots, v_{k_{\mathtt{c}_i}})\ \{\mathtt{c}_j(\boldsymbol{x}) \Rightarrow \boldsymbol{u_j}\}\rangle\ E \\ \to_{\mathtt{case}} C\langle u_i\rangle\ [x_1{\leftarrow}v_1] \ldots [x_{k_{\mathtt{c}_i}}{\leftarrow}v_{k_{\mathtt{c}_i}}] :: E \end{array}$$

As before, a derivation $d : t \to^*_{\mathtt{CbV}} u$ is a potentially empty sequence of evaluation steps, whose length / number of $\to_a$ steps is denoted by $|d|$ / $|d|_a$ for $a \in \{\mathtt{m}, \mathtt{e}, \mathtt{case}, \mathtt{err}_1, \mathtt{err}_2\}$.

The free and bound variables of a term are defined as expected—$\mathtt{err}$ has no free variables. The free variables of a program are defined by looking at environments from the end, as follows:

$$\mathtt{fv}(t, \epsilon) := \mathtt{fv}(t) \qquad \mathtt{fv}(t, E :: [x{\leftarrow}v]) := (\mathtt{fv}(t, E) \setminus \{x\}) \cup \mathtt{fv}(v)$$

As expected, a program is *closed* if its set of free variables is empty. As it is standard, we silently work modulo α-equivalence.

Progress and harmony. The choice of LIME for our study is justified by the similarity with the formalisms used in the studies on functional cost models [6,9,11,4] and with the one used in the Coq abstract machine designed by Barras [16]. A further justification is the fact that it is conservative with respect to CbV closed λ-calculus in a sense that we are now going to explain.

A fundamental property of the CbV closed λ-calculus is that terms either evaluate to a value or they diverge. This property has been highlighted and called *progress* by Wright and Felleisen [39] and later extensively used by Pierce [33], among others. In these studies, however, the property is studied in relationship to a typing system, as a tool to prove its soundness (*typed programs cannot go wrong*). Accattoli and Guerrieri in [11] focus on it in an *untyped* setting and call it *harmony* because it expresses a form of internal completeness, in two ways. First, it shows that in the closed λ-calculus CbV can be seen as a notion of *call-by-normal-form*. Note the subtlety: one cannot define call-by-normal-form evaluation directly, because one needs evaluation to define normal forms—a call-by-normal-form calculus thus requires a certain harmony in its definition. Second, the property shows that the restriction to CbV β-reduction has an impact on the order in which redexes are evaluated, but evaluation never gets stuck, as every β-redex will eventually become a CbV β-redex and be fired, unless evaluation diverges (and with no need of types). In [11], harmony is showed to hold for the *fireball calculus*, an extension of the CbV closed λ-calculus with open terms. LIME rests on closed terms but adds constructors and pattern matching, and so its harmony does not follow from the one of the closed λ-calculus.

Now, we show that LIME is harmonious—types have no role here, so we prefer to refer to *harmony* rather than to *progress*. Let us stress however that

harmony has no role in the complexity analysis, it is presented here only to show that LIME is not ad-hoc.

Harmony is generally showed for single steps, showing that a term either reduces or it is a value.

Proposition 2 (Progress / harmony for LIME). *Let (t, E) be a closed program. Then either $(t, E) \to_{\mathtt{CbV}} (u, E')$ or t is a value.*

Proof. By induction on t. Cases:

- *Value*, *i.e.* $t = v$. Then no rules apply;
- *Variable*, *i.e.* $t = x$. Note that E contains a substitution $[x \leftarrow v]$ because the program is closed, and so $\to_{\mathtt{e}}$ applies.
- *Application*, *i.e.* $t = u\ s$. The *i.h.* on (s, E) gives
  - *s reduces.* Then so does (t, E);
  - *s is a value.* The *i.h.* on (u, E) gives
    - *u reduces.* Then so does (t, E);
    - *u is a value.* Then either $\to_{\mathtt{m}}$ (if u is an abstraction) or $\to_{\mathtt{err}_2}$ (if u is a constructor) applies.
- *Constructor that is not a value*, *i.e.* $t = \mathtt{c}(\boldsymbol{u})$. Then there is a rightmost argument s in $\boldsymbol{u}$ that is not a value. By *i.h.*, (s, E) reduces, and so does $(\mathtt{c}(\boldsymbol{u}), E)$.
- *Match*, *i.e.* $t = \mathtt{case}\, u\, \{b\}$. The *i.h.* on (u, E) gives:
  - *u reduces.* Then so does (t, E);
  - *u is a constructor.* Then $\to_{\mathtt{case}}$ applies;
  - *u is an abstraction.* Then $\to_{\mathtt{err}_1}$ applies. □

Complexity analysis. For complexity analyses, one usually assumes that the initial program p comes with an empty environment, that is, $p = (t_0, \epsilon)$. The two fundamental parameters for analyses of a derivation $d : (t_0, \epsilon) \to^*_{\mathtt{CbV}} q$ are

1. *Length of its small-step evaluation*: the number $|d|_{\mathtt{m}}$ of $\mathtt{m}$-steps in the derivation d, that morally is the number of β-steps at the omitted small-step level.
2. *Input*: the size $|t_0|$ of the initial term t_0;

Our aim is to show that the length $|d|$ of a d is bilinear, that is, linear in $|d|_{\mathtt{m}}$ and $|t_0|$. Since error-handling rules can only appear once, and only at the end of a derivation, they do not really play a role. Therefore, the goal is to prove that the number of exponential $\to_{\mathtt{e}}$ and matching $\to_{\mathtt{case}}$ steps is bilinear. To prove it, we need the following measure $|\cdot|_{\mathtt{v}}$ of terms and programs (where k is the number of constructors in the language and $k_{\mathtt{c}_i}$ is the arity of the i-th constructor), that simply counts the number of free variable occurrences and of `case` constructs out of abstractions, *i.e.* of the locations where $\to_{\mathtt{e}}$ and $\to_{\mathtt{case}}$ steps can act:

$$|x|_{\mathtt{v}} := 1 \qquad |\lambda x.t|_{\mathtt{v}} := 0 \qquad |\mathtt{err}|_{\mathtt{v}} := 0$$
$$|t\ u|_{\mathtt{v}} := |t|_{\mathtt{v}} + |u|_{\mathtt{v}} \qquad |\mathtt{c}_i(\boldsymbol{t})|_{\mathtt{v}} := \Sigma_{j=1}^{k_{\mathtt{c}_i}} |t_j|_{\mathtt{v}} \qquad |(t, E)|_{\mathtt{v}} := |t|_{\mathtt{v}}$$
$$|\mathtt{case}\, t\, \{\mathtt{c}_i(\boldsymbol{x}) \Rightarrow \boldsymbol{u_i}\}|_{\mathtt{v}} := 1 + |t|_{\mathtt{v}} + max\{|u_i|_{\mathtt{v}} \mid i = 1, \ldots, k\}$$

Note that for the branches of a `case` construct we use the *max*, because only one of them is selected by $\to_{\texttt{case}}$ while the others are discarded. The measure is extended to evaluation contexts by setting $|\langle\cdot\rangle|_{\texttt{v}} := 0$ and defining it on the other cases as for terms. The following properties of the measure follow immediately from the definition:

Lemma 3 (Basic properties of the measure).

1. Values*:* $|v|_{\texttt{v}} = 0$ *for every value* v.
2. Size Upper Bound*:* $|t|_{\texttt{v}} \leq |t|$ *for every term* t.
3. Context Factorization*:* $|C\langle t\rangle|_{\texttt{v}} = |C|_{\texttt{v}} + |t|_{\texttt{v}}$.

From these properties a straightforward inspection of the rules shows, as expected, that

Lemma 4 (Exponential and matching rules decrease the measure). *If* $(t, E) \to_a (u, E')$ *then* $|(t, E)|_{\texttt{v}} > |(u, E')|_{\texttt{v}}$ *for* $a \in \{\texttt{e}, \texttt{case}\}$.

Lemma 4 implies that the length of a sequence of exponential and matching steps is bounded by the measure of the code at the beginning of the sequence, that by Lemma 3.2 is bounded by the size of that code. To conclude, we have to establish the connection between multiplicative steps and code sizes. It turns out that $\to_{\texttt{m}}$ can increase the measure only by an amount bounded by the size of the initial term. This property follows by an invariant known as *the subterm property*, that relates the size of terms along the derivation with the size of the initial one. It is the key property for complexity analyses, playing a role akin to that of the cut-elimination theorem for sequent calculi, or of the subformula property for proof search. It does not hold in the ordinary λ-calculus, because it requires meta-level substitution to be decomposed in micro-steps. It can instead be found in many abstract machines and other setting decomposing β-reduction.

The subterm property can be formulated in various ways. Sometimes it states that the size of duplicated subterms is bounded by the size of the initial term. In LIME, it takes a different form. The multiplicative rule $\to_{\texttt{m}}$ can increase the measure because it opens an abstraction, that being a value has measure 0, and potentially exposes new free variable occurrences and `case` constructs. Therefore, the important point is to bound the size of abstraction bodies, which is why the property takes the following form.

Lemma 5 (LIME subterm property). *Let* $d : (t_0, \epsilon) \to^*_{\texttt{CbV}} (u, E)$ *be a LIME derivation. Then the size of every abstraction in* u *and* E *is bounded by the size* $|t_0|$ *of the initial term.*

Proof. By induction on the length of the derivation d. The base case $|d| = 0$ is immediate. For a non-empty derivation consider the last step $(s, E') \to_{\texttt{CbV}} (u, E)$. By *i.h.*, the statement holds for (s, E'). The rules may move abstractions from s to E' or vice-versa, but they never substitute inside abstractions (evaluation contexts are weak, *i.e.* they do not go under abstraction) nor create them out of the blue. □

Let us stress why the property requires meta-level substitution to be decomposed: it is only because LIME never replaces variable occurrences under abstraction that the size of abstractions does not grow.

We can then conclude with the bound on the length of derivations.

Theorem 6 (LIME bilinear bound). *Let $d : (t_0, \epsilon) \to^*_{\mathtt{CbV}} (u, E)$ be a LIME derivation. Then $|d| = O(|t_0| \cdot (|d|_{\mathtt{m}} + 1))$.*

Proof. First of all, note that a error-handling rules can appear only at the end of the evaluation process, and they end it. So, we omit them, and consider them included in the *big O* notation, in the additive constant.

The measure is non-negative, and at the beginning is bound by the size $|t_0|$ of the initial term, by the size upper bound (Lemma 3.2). Rules $\to_{\mathtt{e}}$ and $\to_{\mathtt{case}}$ decrease the size, that is increased only by the multiplicative rule $\to_{\mathtt{m}}$ that opens an abstraction (whose content was ignored by the measure before) but the increment given by the body of the abstraction is bound by the size of the initial term by the subterm property (plus the size upper bound and the context factorization of the measure). Thus the number of $\to_{\mathtt{e}}$ and $\to_{\mathtt{case}}$ steps is bound by $|t_0| \cdot (|d|_{\mathtt{m}} + 1)$. Finally, one has to add the multiplicative steps themselves, and the eventual final error step—therefore, $|d| = O(|t_0| \cdot (|d|_{\mathtt{m}} + 1))$. □

Tightness of the bilinear bound, and the increased number of exponentials. We finish this study by showing that this bound is asymptotically optimal, that is, by showing a family of derivations reaching the bilinear bound. Our family is a diverging one, obtained by a simple hack of the famous diverging term $\delta\delta$. Of course, the example can be made terminating at the cost of some additional technicalities, we use a diverging family only for the sake of simplicity.

Before giving the example, let us point out a subtlety. Theorem 6 states in particular that the number of exponential steps is bilinear. Accattoli and Sacerdoti Coen have shown that in the CbV (and CbNeed) closed λ-calculus (that is, without pattern matching) a stronger bound holds: exponentials do not depend on the size of the initial term, and are linear only in the number of β-steps. It is natural to wonder whether in LIME the bilinearity involves only matching steps, and so exponentials are actually linear, or if instead both matching and exponential steps are bilinear. The example shows that both are bilinear.

For the example, we consider a unary constructor $\mathtt{c}$ and a zeroary constructor 0, but for the sake of conciseness the matching constructs in the family will specify only one branch. Define:

$$\begin{array}{rlrl} C_0 &:= (y\ y)\ \langle\cdot\rangle & \qquad \delta_n &:= \lambda y.\lambda x_n.C_n\langle x_n\rangle \\ C_{n+1} &:= \mathtt{case}\ \mathtt{c}(\langle\cdot\rangle)\ \{\mathtt{c}(x_n) \Rightarrow C_n\langle x_n\rangle\} & \qquad t_n &:= (\delta_n\ \delta_n)\ 0 \end{array}$$

Note that

$$\begin{array}{rcl} (C_n\langle 0\rangle, E) & = & (\mathtt{case}\ \mathtt{c}(0)\ \{\mathtt{c}(x_{n-1}) \Rightarrow C_{n-1}\langle x_{n-1}\rangle\}, E) \\ & \to_{\mathtt{case}} & (C_n\langle x_{n-1}\rangle, [x_{n-1}{\leftarrow}0] :: E) \\ & \to_{\mathtt{e}} & (C_{n-1}\langle 0\rangle, [x_{n-1}{\leftarrow}0] :: E) \end{array}$$

And so we can iterate, obtaining the derivation:

$$\begin{array}{rcl}(C_n\langle 0\rangle, E) & = & (\mathtt{case}\,\mathtt{c}(0)\,\{\mathtt{c}(x_{n-1}) \Rightarrow C_{n-1}\langle x_{n-1}\rangle\}, E)\\ & (\to_{\mathtt{case}}\to_{\mathtt{e}})^n & (C_0\langle 0\rangle, [x_0{\leftarrow}0] :: \ldots :: [x_{n-1}{\leftarrow}0] :: E)\end{array}$$

Defining $E_0 := [x_0{\leftarrow}0] :: \ldots :: [x_{n-1}{\leftarrow}0]$ we then have $(C_n\langle 0\rangle, E)\ (\to_{\mathtt{case}}\to_{\mathtt{e}})^n\ (C_0\langle 0\rangle, E_0 :: E)$. Starting from t_n and a generic environment E, we obtain the following derivation d (that does not in fact depend on E):

$$\begin{array}{rcl}(t_n, E) & = & (((\lambda y.\lambda x_n.C_n\langle x_n\rangle)\ \delta_n)\ 0, E)\\ & \to_{\mathtt{m}} & ((\lambda x_n.C_n\langle x_n\rangle)0, [y{\leftarrow}\delta_n] :: E)\\ & \to_{\mathtt{m}} & (C_n\langle x_n\rangle, [x_n{\leftarrow}0] :: [y{\leftarrow}\delta_n] :: E)\\ & \to_{\mathtt{e}} & (C_n\langle 0\rangle, [x_n{\leftarrow}0] :: [y{\leftarrow}\delta_n] :: E)\\ & (\to_{\mathtt{case}}\to_{\mathtt{e}})^n & (C_0\langle 0\rangle, E_0 :: [x_n{\leftarrow}0] :: [y{\leftarrow}\delta_n] :: E)\\ & = & ((y\ y)0, E_0 :: [x_n{\leftarrow}0] :: [y{\leftarrow}\delta_n] :: E)\\ & \to_{\mathtt{e}} & ((y\ \delta_n)0, E_0 :: [x_n{\leftarrow}0] :: [y{\leftarrow}\delta_n] :: E)\\ & \to_{\mathtt{e}} & ((\delta_n\ \delta_n)0, E_0 :: [x_n{\leftarrow}0] :: [y{\leftarrow}\delta_n] :: E)\\ & = & (t_n, E_0 :: [x_n{\leftarrow}0] :: [y{\leftarrow}\delta_n] :: E)\end{array}$$

More compactly, $(t_n, E) \to^*_{\mathtt{CbV}} (t_n, E')$ with $O(1)$ (namely 2) $\to_{\mathtt{m}}$ steps and $\Omega(n)$ $\to_{\mathtt{case}}$ and $\Omega(n)$ $\to_{\mathtt{e}}$ steps. Now, consider the m-th iteration d^m of d starting from (t_n, ϵ). Since the size of the initial term is proportional to n (*i.e.* $|t_n| = \Theta(n)$), the number of steps in d^m is linear in the size of the initial term t_n, and each iteration is enabled by a $\beta/\mathtt{m}$ step, so it is also linear in the number of β-steps. That is, we obtained that both $|d^m|_{\mathtt{e}}$ and $|d^m|_{\mathtt{case}}$ have lower bound $\Omega(|t_n| \cdot |d^m|_{\mathtt{m}})$, reaching the bilinear upper bound for both kinds of step.

4 Call-by-Need, LINED, and the Bilinear Bound

CbNeed evaluation is the variation over CbV where arguments that are not needed are not evaluated, so that the cases in which CbV diverges but CbN terminates are avoided, marrying the efficiency of CbV with the better behavior with respect to termination of CbN—classic references on CbNeed are [37,30,31,15,36].

Being based on CbV, CbNeed rests on values, and for our study the key point turns out to be the definition of values in the case of constructors. In this section constructors are values only when their arguments are variables. Under this hypothesis, we can smoothly adapt the proof of the previous section, and show that pattern matching is negligible. In the next section we shall study the variant in which every constructor is considered as a value, independently of the shape of its arguments.

Here we adopt the presentation of CbNeed of Accattoli, Barenbaum, and Mazza [6], resting on the linear substitution calculus. With respect to LIME, the only difference is that the environment is integrated inside the term itself and the notion of program disappears—in CbNeed is not possible to disentangle the term and the environment, unless more data structures are used. Let us call this framework *LINED*, for *LInear matching calculus by NEeD*.

The grammar of LINED is:

$$
\begin{array}{rl}
\textsc{Terms} & t,u ::= x \mid \lambda x.t \mid t\ u \mid \mathtt{c}(\boldsymbol{t}) \mid \mathtt{case}\, t\, \{b\} \mid \mathtt{err} \mid t[x{\leftarrow}u] \\
\textsc{Branches} & b ::= \mathtt{c_i}(\boldsymbol{x}) \Rightarrow \boldsymbol{u_i} \\
\textsc{Values} & v,w ::= \lambda x.t \mid \mathtt{c}(\boldsymbol{x}) \mid \mathtt{err} \\
\\
\textsc{Subs. Contexts} & L ::= \langle\cdot\rangle \mid L[x{\leftarrow}t] \\
\textsc{Eval. Contexts} & N,M ::= \langle\cdot\rangle \mid N\ t \mid N[x{\leftarrow}u] \mid M\langle x\rangle[x{\leftarrow}N] \mid \mathtt{case}\, N\, \{b\} \\
\textsc{Answers} & a ::= L\langle v\rangle
\end{array}
$$

where $t[x{\leftarrow}u]$ is called an *explicit substitution* (ES) and binds x in t—it is absolutely equivalent to write $\mathtt{let}\ x = u\ \mathtt{in}\ t$, it is just more concise. Note the category of answers, that are simply values in an environment.

The key point for CbNeed evaluation is the case $M\langle x\rangle[x{\leftarrow}N]$ in the definition of evaluation contexts (where we implicitly assume that M does not bind x), whose role is to move evaluation inside the ES / environment $[x{\leftarrow}N]$.

Rewriting rules. Now that the environment is entangled with the term, most rules have to work up to a segment of the environment, that is, a substitution context L. This is standard in the framework of the linear substitution calculus. All rules but the last one ($\to_{\mathtt{err}_3}$, that is a global rule) are defined at top level and then closed by evaluation contexts:

$$
\begin{array}{rcll}
\multicolumn{4}{c}{\textsc{Rules at Top Level (plus } \to_{\mathtt{err}_3}\textsc{)}} \\
L\langle\lambda x.t\rangle\ u & \mapsto_{\mathtt{m}} & L\langle t[x{\leftarrow}u]\rangle & \\
N\langle x\rangle[x{\leftarrow}L\langle v\rangle] & \mapsto_{\mathtt{e}} & L\langle N\langle v\rangle[x{\leftarrow}v]\rangle & \\
\mathtt{case}\, L\langle \mathtt{c}_i(\boldsymbol{y})\rangle\, \{\mathtt{c_i}(\boldsymbol{x}) \Rightarrow \boldsymbol{u_i}\} & \mapsto_{\mathtt{case}} & L\langle u_i[\boldsymbol{x}{\leftarrow}\boldsymbol{y}]\rangle & \\
\mathtt{c}(\boldsymbol{t}) & \mapsto_{\mathtt{cstr}} & \mathtt{c}(\boldsymbol{x})[\boldsymbol{x}{\leftarrow}\boldsymbol{t}] & \text{if } \boldsymbol{t} \neq \boldsymbol{y} \\
\mathtt{case}\, L\langle\lambda x.t\rangle\, \{b\} & \mapsto_{\mathtt{err}_1} & \mathtt{err} & \\
L\langle \mathtt{c}(\boldsymbol{x})\rangle\ t & \mapsto_{\mathtt{err}_2} & \mathtt{err} & \\
N\langle \mathtt{err}\rangle & \to_{\mathtt{err}_3} & \mathtt{err} &
\end{array}
$$

$$
\begin{array}{c}
\textsc{Contextual closure} \\
N\langle t\rangle \to_a N\langle u\rangle \quad \text{if } t \mapsto_a u \qquad \text{for } a \in \{\mathtt{m}, \mathtt{e}, \mathtt{cstr}, \mathtt{case}, \mathtt{err}_1, \mathtt{err}_2\}
\end{array}
$$

We use $\to_{\mathtt{CbNeed}}$ to denote the union of all these rules. Note the side condition $\boldsymbol{t} \neq \boldsymbol{y}$ for $\mapsto_{\mathtt{cstr}}$: it is a compact way of saying that at least one term in $\boldsymbol{t}$ is not a variable, whose aim is to avoid silly diverging derivations. The rule can be optimized by avoiding to replace those elements in $\boldsymbol{t}$ that are already variables, but to show that the overhead is not exponential this is not needed. Note also that rule $\to_{\mathtt{case}}$ now asks the arguments of the constructor to match to be variables, because if they are not then $\to_{\mathtt{cstr}}$ applies first.

Harmony. As for LIME, harmony holds for LINED, and, as before, we show it to stress that LINED is not an ad-hoc framework. Here, however, it is formulated in a slightly different way, on open terms. The reason is that in the case of a term of the form $t[x{\leftarrow}u]$, the subterm t—to which we want to apply the inductive hypothesis—might be open even when the whole term is closed. Therefore harmony has now a new, third case for open terms: closed terms however cannot fall in this category, and so on them harmony takes its usual form.

Proposition 7 (Progress / harmony for LINED). *Let t be a term of LINED. Either $t \to_{\mathtt{CbNeed}} u$, or t is an answer, or t is an open term of the form $N\langle x\rangle$ where N does not bind x.*

Proof. By induction on t. Cases:

- *Value*, *i.e.* $t = v$. Then t is an answer (and it is not of the two other forms).
- *Variable*, *i.e.* $t = x$. Then t is an open term.
- *Application*, *i.e.* $t = u\ s$. The *i.h.* on u gives
  - u *reduces or is open.* Then so does t;
  - u *is a constructor value in a substitution context.* Then $\to_{\mathtt{err}_2}$ applies;
  - u *is an abstraction in a substitution context.* Then $\to_{\mathtt{m}}$ applies.
  - u *is an error in a substitution context.* Then $\to_{\mathtt{err}_3}$ applies.
- *Substitution*, *i.e.* $t = u[x \leftarrow s]$. The *i.h.* on u gives
  - u *reduces or is open with head variable not* x. Then so does t;
  - u *is open with hereditary head variable* x. Then $\to_{\mathtt{e}}$ applies;
  - u *is an answer.* Then so is t.
- *Constructor that is not a value*, *i.e.* $t = \mathtt{c}(\boldsymbol{u})$. Then $\to_{\mathtt{cstr}}$ applies.
- *Match*, *i.e.* $t = \mathtt{case}\, u\, \{b\}$. The *i.h.* on u gives:
  - u *reduces or is open.* Then so does t;
  - u *is a constructor value in a substitution context.* Then $\to_{\mathtt{case}}$ applies;
  - u *is an abstraction in a substitution context.* Then $\to_{\mathtt{err}_1}$ applies.
  - u *is an error in a substitution context.* Then $\to_{\mathtt{err}_3}$ applies. □

Complexity analysis. The bounds for LINED are obtained following the same reasoning done for LIME, but using a slightly different measure. There are two differences. First, in LINED evaluation enters inside ES, so now the measure takes them into account. Second, in LINED the analysis has to bound also the number of $\to_{\mathtt{cstr}}$ steps, not present in LIME. Accordingly, the measure now counts 1 for every constructor out of abstractions. The measure $|\cdot|_{\mathtt{n}}$ for LINED is thus defined by:

$$|x|_{\mathtt{n}} := 1 \qquad |v|_{\mathtt{n}} := 0 \qquad |t[x \leftarrow u]|_{\mathtt{n}} := |t|_{\mathtt{n}} + |u|_{\mathtt{n}}$$

$$|t\ u|_{\mathtt{n}} := |t|_{\mathtt{n}} + |u|_{\mathtt{n}} \qquad |\mathtt{c}_i(\boldsymbol{t})|_{\mathtt{n}} := 1 + \Sigma_{j=1}^{k_{\mathtt{c}_i}} |t_j|_{\mathtt{n}}$$

$$|\mathtt{case}\, t\, \{\mathtt{c}_{\boldsymbol{i}}(\boldsymbol{x}) \Rightarrow \boldsymbol{u_i}\}|_{\mathtt{n}} := 1 + |t|_{\mathtt{n}} + max\{k_{\mathtt{c}_i} + |u_i|_{\mathtt{n}} \mid i = 1, \ldots, k\}$$

Note that also the definition on `case` constructs is different with respect to the measure for LIME, as it now adds $k_{\mathtt{c}_i}$. The reason: $\to_{\mathtt{case}}$ creates $k_{\mathtt{c}_i}$ ES that in LINED contribute to the measure, while in LIME they do not.

As before, the measure is extended to evaluation contexts by setting $|\langle\cdot\rangle|_{\mathtt{v}} := 0$ and defining it on the other cases as for terms. The following properties of the measure follow immediately from the definition:

Lemma 8 (Basic properties of the measure).

1. Size Upper Bound*:* $|t|_{\mathtt{n}} \leq |t|$ *for every term* t.
2. Context Factorization*:* $|N\langle t\rangle|_{\mathtt{n}} = |N|_{\mathtt{n}} + |t|_{\mathtt{n}}$ *and in particular* $|L\langle t\rangle|_{\mathtt{n}} = |L|_{\mathtt{n}} + |t|_{\mathtt{n}}$.

Next, we show that the measure decreases with the rules other than the multiplicative one, standing for β, and the error handling rules (that are trivial).

Lemma 9 (Exponential, matching and constructor rules decrease the measure). *If $t \to_a u$ then $|t|_{\mathtt{n}} > |u|_{\mathtt{n}}$ for $a \in \{\mathtt{e}, \mathtt{case}, \mathtt{cstr}\}$.*

Proof.

- *Exponential*: $t = N\langle x\rangle[x{\leftarrow}L\langle v\rangle] \to_{\mathtt{e}} L\langle N\langle v\rangle[x{\leftarrow}v]\rangle = u$. Then:

$$\begin{aligned}|N\langle x\rangle[x{\leftarrow}L\langle v\rangle]|_{\mathtt{n}} &= 1 + |N|_{\mathtt{n}} + 0 + |L|_{\mathtt{n}} > \\ &\quad 0 + |N|_{\mathtt{n}} + 0 + |L|_{\mathtt{n}} = |L\langle N\langle v\rangle[x{\leftarrow}v]\rangle|_{\mathtt{n}}\end{aligned}$$

- *Matching*: $t = \mathtt{case}\, L\langle \mathtt{c}_i(\boldsymbol{y})\rangle\, \{\mathtt{c}_{\boldsymbol{i}}(\boldsymbol{x}) \Rightarrow \boldsymbol{u_i}\} \to_{\mathtt{case}} L\langle u_i[\boldsymbol{x}{\leftarrow}\boldsymbol{y}]\rangle = u$. Then:

$$\begin{aligned}|t|_{\mathtt{n}} &= 1 + 0 + |L|_{\mathtt{n}} + max\{k_{\mathtt{c}_j} + |u_j|_{\mathtt{n}} \mid j = 1, \dots, k\} \\ &> |L|_{\mathtt{n}} + k_{\mathtt{c}_i} + |u_i|_{\mathtt{n}} \qquad = |L\langle u_i[\boldsymbol{x}{\leftarrow}\boldsymbol{y}]\rangle|_{\mathtt{n}}\end{aligned}$$

- *Constructor*: $t = \mathtt{c}(\boldsymbol{t}) \to_{\mathtt{cstr}} \mathtt{c}(\boldsymbol{x})[\boldsymbol{x}{\leftarrow}\boldsymbol{t}] = u$. We have:

$$|\mathtt{c}(\boldsymbol{t})|_{\mathtt{n}} = 1 + \Sigma|\boldsymbol{t}|_{\mathtt{n}} > 0 + \Sigma|\boldsymbol{t}|_{\mathtt{n}} = |\mathtt{c}(\boldsymbol{x})[\boldsymbol{x}{\leftarrow}\boldsymbol{t}]|_{\mathtt{n}} \quad \square$$

As for LIME, the bilinear bound rests on a subterm property. Both the property and the bound are proved exactly as in the CbV case. Moreover, the example showing that the bound for LIME is tight applies also to LINED.

Lemma 10 (LINED subterm property). *Let $d : t_0 \to^*_{\mathtt{CbNeed}} u$ be a LINED derivation. Then the size of every abstraction in u and is bounded by the size $|t_0|$ of the initial term.*

Theorem 11 (LINED bilinear bound). *Let $d : t_0 \to^*_{\mathtt{CbNeed}} u$ be a LINED derivation. Then $|d| = O(|t_0| \cdot (|d|_{\mathtt{m}} + 1))$.*

5 Call-by-Need, ExpLINED, and Matching Explosion

Here we consider ExpLINED, a variant of LINED where constructors are always considered as values, not only when they are applied to variables. The effect of this change is dramatic: it re-introduces matching explosions, even if arguments are still evaluated once and for all, because constructors then can be exploited to block the evaluation of subterms. This case study is used to stress two facts: first, the no negligible cost hypothesis for pattern matching is less obvious than it seems, and second, the study of cost models can be used as a language design principle, to discriminate between different and yet equivalent operational semantics[2].

[2] We do not prove the equivalence between the two formulations of CbNeed studied in the paper, but the difference is essentially that in one case $\mathtt{c}(\boldsymbol{t})$ is reduced to $\mathtt{c}(\boldsymbol{x})[\boldsymbol{x}{\leftarrow}\boldsymbol{t}]$ (via $\to_{\mathtt{cstr}}$) while in the other case it is left unchanged—the two calculi compute the same result, up to substitutions, just with very different complexities.

ExpLINED. For the sake of conciseness and readability, ExpLINED is defined by pointing out the differences with respect to LINED, rather than repeating all the definitions. The grammar of values of ExpLINED is:

$$v ::= \lambda x.t \mid \mathtt{err} \mid \mathtt{c}(\boldsymbol{t})$$

Dynamically, rule $\to_{\mathtt{cstr}}$ is removed while $\to_{\mathtt{case}}$ is slightly modified, to fire with every constructor, independently of the shape of its arguments:

$$\mathtt{case}\, L\langle \mathtt{c}_i(\boldsymbol{t})\rangle\, \{\mathtt{c}_{\boldsymbol{i}}(\boldsymbol{x}) \Rightarrow \boldsymbol{u_i}\} \to_{\mathtt{case}} L\langle u_i[\boldsymbol{x}\leftarrow\boldsymbol{t}]\rangle$$

Matching exploding family. We are now going to define a matching exploding family. The idea is similar to that of the family for CbN, that is, to repeatedly trigger the evaluation of arguments—in CbN we used arguments of β-redexes, now we exploit constructor arguments. The family is trickier to define and analyze. In fact, the definition of the family requires a delicate decomposition via contexts, and the calculations are more involved. Moreover, it took us a lot more time to find it. The trick, however, is essentially the same used for CbN.

As before, we use two constructors $\mathtt{c}$, that is unary, and 0, that is zeroary. We introduce various notions of contexts, and the exploding family is given by $D_n\langle t_n\rangle$, but we decompose the analysis in two steps.

Terms and contexts are then defined by:

$$\begin{array}{l} E_n := \mathtt{case}\, x_n\, \{\mathtt{c}(y) \Rightarrow \mathtt{case}\, y\, \{0 \Rightarrow \langle\cdot\rangle\}\} \\ t_n := E_n\langle E_n\langle 0\rangle\rangle \end{array}$$

$$\begin{array}{rlrl} C_1 &:= \langle\cdot\rangle[x_1\leftarrow\mathtt{c}(0)] & \qquad D_1 &:= (\lambda x_1.\langle\cdot\rangle)\mathtt{c}(0) \\ C_{n+1} &:= C_n\langle\langle\cdot\rangle[x_{n+1}\leftarrow\mathtt{c}(t_n)]\rangle & \qquad D_{n+1} &:= D_n\langle(\lambda x_{n+1}.\langle\cdot\rangle)\mathtt{c}(t_n)\rangle \end{array}$$

Proposition 12.

1. Linear multiplicative prefix*: for any term u there exists a derivation $d_n : D_n\langle u\rangle \to_{\mathtt{m}}^n C_n\langle u\rangle$;*
2. Exponential pattern matching suffix*: if N does not capture x_n then there exists a context L and a derivation $e_n : C_n\langle N\langle t_n\rangle\rangle \to^* C_n\langle N\langle L\langle 0\rangle\rangle\rangle$ with $|e_n|_{\mathtt{case}} = \Omega(2^{n+1})$ and $|e_n|_{\mathtt{e}} = \Omega(2^{n+1})$.*
3. Matching exploding family*: there exists a context L and a derivation $f_n : D_n\langle t_n\rangle \to^* C_n\langle L\langle 0\rangle\rangle$ with $|D_n\langle t_n\rangle| = O(n)$, $|f_n|_{\mathtt{m}} = n$, $|f_n|_{\mathtt{case}} = \Omega(2^{n+1})$, and $|f_n|_{\mathtt{e}} = \Omega(2^{n+1})$.*

Proof. Point 3 is obtained by concatenating Point 1 and Point 2 (taking the empty evaluation context $N = \langle\cdot\rangle$).

Point 1 and Point 2 are by induction on n. Cases:

- *Base case, i.e. $n = 1$.*
  1. *Linear multiplicative prefix*: the derivation d_1 is given by

$$\begin{array}{rcl} D_1\langle u\rangle &=& (\lambda x_1.u)\mathtt{c}(0) \\ &\to_{\mathtt{m}}& u[x_1\leftarrow\mathtt{c}(0)] = C_1\langle u\rangle \end{array}$$

2. *Exponential pattern matching suffix*: the first part of the evaluation e_1 of the statement is given by

$$\begin{array}{rcl} C_1\langle N\langle t_1\rangle\rangle & = & N\langle t_1\rangle[x_1{\leftarrow}\mathtt{c}(0)] \\ & = & N\langle \mathtt{case}\, x_1\, \{\mathtt{c}(y) \Rightarrow \mathtt{case}\, y\, \{0 \Rightarrow E_n\langle 0\rangle\}\}\rangle[x_1{\leftarrow}\mathtt{c}(0)] \\ & \to_{\mathtt{e}} & N\langle \mathtt{case}\, \mathtt{c}(0)\, \{\mathtt{c}(y) \Rightarrow \mathtt{case}\, y\, \{0 \Rightarrow E_n\langle 0\rangle\}\}\rangle[x_1{\leftarrow}\mathtt{c}(0)] \\ & \to_{\mathtt{case}} & N\langle \mathtt{case}\, y\, \{0 \Rightarrow E_n\langle 0\rangle\}[y{\leftarrow}0]\rangle[x_1{\leftarrow}\mathtt{c}(0)] \\ & \to_{\mathtt{e}} & N\langle \mathtt{case}\, 0\, \{0 \Rightarrow E_n\langle 0\rangle\}[y{\leftarrow}0]\rangle[x_1{\leftarrow}\mathtt{c}(0)] \\ & \to_{\mathtt{case}} & N\langle E_n\langle 0\rangle[y{\leftarrow}0]\rangle[x_1{\leftarrow}\mathtt{c}(0)] \end{array}$$

Let us now expand E_n and continue with the second part of e_1:

$$\begin{array}{rcl} & & N\langle E_n\langle 0\rangle[y{\leftarrow}0]\rangle[x_1{\leftarrow}\mathtt{c}(0)] \\ & = & N\langle \mathtt{case}\, x_1\, \{\mathtt{c}(z) \Rightarrow \mathtt{case}\, z\, \{0 \Rightarrow 0\}\}[y{\leftarrow}0]\rangle[x_1{\leftarrow}\mathtt{c}(0)] \\ & \to_{\mathtt{e}} & N\langle \mathtt{case}\, \mathtt{c}(0)\, \{\mathtt{c}(z) \Rightarrow \mathtt{case}\, z\, \{0 \Rightarrow 0\}\}[y{\leftarrow}0]\rangle[x_1{\leftarrow}\mathtt{c}(0)] \\ & \to_{\mathtt{case}} & N\langle \mathtt{case}\, z\, \{0 \Rightarrow 0\}[z{\leftarrow}0][y{\leftarrow}0]\rangle[x_1{\leftarrow}\mathtt{c}(0)] \\ & \to_{\mathtt{e}} & N\langle \mathtt{case}\, 0\, \{0 \Rightarrow 0\}[z{\leftarrow}0][y{\leftarrow}0]\rangle[x_1{\leftarrow}\mathtt{c}(0)] \\ & \to_{\mathtt{case}} & N\langle 0[z{\leftarrow}0][y{\leftarrow}0]\rangle[x_1{\leftarrow}\mathtt{c}(0)] \\ & = & N\langle L\langle 0\rangle\rangle[x_1{\leftarrow}\mathtt{c}(0)] \\ & = & C_1\langle N\langle L\langle 0\rangle\rangle\rangle \end{array}$$

where $|e_1|_{\mathtt{case}} = 4 = \Omega(2^{1+1})$ and $|e_1|_{\mathtt{e}} = 4 = \Omega(2^{1+1})$.

– *Inductive case.*

1. *Linear multiplicative prefix*: note that C_n is an evaluation context for every n. Then d_{n+1} is given by

$$\begin{array}{rcl} D_{n+1}\langle u\rangle & = & D_n\langle(\lambda x_{n+1}.u)\mathtt{c}(t_n)\rangle \\ (\text{by } \textit{i.h.}) & \to_{\mathtt{m}}^{d_n} & C_n\langle(\lambda x_{n+1}.u)\mathtt{c}(t_n)\rangle \\ & \to_{\mathtt{m}} & C_n\langle u[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \quad = C_{n+1}\langle u\rangle \end{array}$$

2. *Exponential pattern matching suffix*: note that $E_n\langle u\rangle$ has the form $N_u\langle x_n\rangle$ with $N_u = \mathtt{case}\, \langle\cdot\rangle\, \{\mathtt{c}(y) \Rightarrow \mathtt{case}\, y\, \{0 \Rightarrow u\}\}$, and so $t_n = E_n\langle E_n\langle 0\rangle\rangle = N_{E_n\langle 0\rangle}\langle x_n\rangle$ and $E_n\langle 0\rangle = N_0\langle x_n\rangle$. The derivation e_{n+1} is constructed as follows. It starts with

$$\begin{array}{rcl} & & C_{n+1}\langle N\langle t_{n+1}\rangle\rangle \\ & = & C_n\langle N\langle t_{n+1}\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\ & = & C_n\langle N\langle N_{E_{n+1}\langle 0\rangle}\langle x_{n+1}\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\ & \to_{\mathtt{e}} & C_n\langle N\langle N_{E_{n+1}\langle 0\rangle}\langle \mathtt{c}(t_n)\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\ & \to_{\mathtt{case}} & C_n\langle N\langle \mathtt{case}\, y\, \{0 \Rightarrow E_{n+1}\langle 0\rangle\}[y{\leftarrow}t_n]\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \end{array}$$

Now, let us set $N' := N\langle \mathtt{case}\, y\, \{0 \Rightarrow E_{n+1}\langle 0\rangle\}[y{\leftarrow}\langle\cdot\rangle]\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]$. Then, e_{n+1} continues as follows

$$\begin{array}{rcl} & & C_n\langle N\langle \mathtt{case}\, y\, \{0 \Rightarrow E_{n+1}\langle 0\rangle\}[y{\leftarrow}t_n]\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\ & = & C_n\langle N'\langle t_n\rangle\rangle \\ (\text{by } \textit{i.h.}) & \to_{e_n}^{*} & C_n\langle N'\langle L\langle 0\rangle\rangle\rangle \\ & = & C_n\langle N\langle \mathtt{case}\, y\, \{0 \Rightarrow E_{n+1}\langle 0\rangle\}[y{\leftarrow}L\langle 0\rangle]\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\ & \to_{\mathtt{e}} & C_n\langle N\langle L\langle \mathtt{case}\, 0\, \{0 \Rightarrow E_{n+1}\langle 0\rangle\}[y{\leftarrow}0]\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\ & \to_{\mathtt{case}} & C_n\langle N\langle L\langle E_{n+1}\langle 0\rangle[y{\leftarrow}0]\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \end{array}$$

Using the equality $E_{n+1}\langle 0\rangle = N_0\langle x_{n+1}\rangle$, we continue with

$$\begin{array}{rl}
 & C_n\langle N\langle L\langle E_{n+1}\langle 0\rangle[y{\leftarrow}0]\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\
= & C_n\langle N\langle L\langle N_0\langle x_{n+1}\rangle[y{\leftarrow}0]\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\
\to_{\mathtt{e}} & C_n\langle N\langle L\langle N_0\langle \mathtt{c}(t_n)\rangle[y{\leftarrow}0]\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\
= & C_n\langle N\langle L\langle \mathtt{case}\,\mathtt{c}(t_n)\,\{\mathtt{c}(z)\Rightarrow \mathtt{case}\,z\,\{0\Rightarrow 0\}\}[y{\leftarrow}0]\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\
\to_{\mathtt{case}} & C_n\langle N\langle L\langle \mathtt{case}\,z\,\{0\Rightarrow 0\}[z{\leftarrow}t_n][y{\leftarrow}0]\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle
\end{array}$$

Now, let us set $N'' := N\langle L\langle \mathtt{case}\,z\,\{0\Rightarrow 0\}[z{\leftarrow}\langle\cdot\rangle][y{\leftarrow}0]\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]$. Then, e_{n+1} continues and ends as follows

$$\begin{array}{rl}
 & C_n\langle N\langle L\langle \mathtt{case}\,z\,\{0\Rightarrow 0\}[z{\leftarrow}t_n][y{\leftarrow}0]\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\
= & C_n\langle N''\langle t_n\rangle\rangle \\
\overset{e_n}{\to}{}^* & C_n\langle N''\langle L'\langle 0\rangle\rangle\rangle \quad \text{(by } i.h.\text{)} \\
= & C_n\langle N\langle L\langle \mathtt{case}\,z\,\{0\Rightarrow 0\}[z{\leftarrow}L'\langle 0\rangle][y{\leftarrow}0]\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\
\to_{\mathtt{e}} & C_n\langle N\langle L\langle L'\langle \mathtt{case}\,0\,\{0\Rightarrow 0\}[z{\leftarrow}0]\rangle[y{\leftarrow}0]\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\
\to_{\mathtt{case}} & C_n\langle N\langle L\langle L'\langle 0[z{\leftarrow}0]\rangle[y{\leftarrow}0]\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\
= & C_n\langle N\langle L''\langle 0\rangle\rangle[x_{n+1}{\leftarrow}\mathtt{c}(t_n)]\rangle \\
= & C_{n+1}\langle N\langle L''\langle 0\rangle\rangle\rangle
\end{array}$$

Now, $|e_{n+1}|_{\mathtt{case}} = 4 + 2\cdot|e_n|_{\mathtt{case}} =_{i.h.} 4 + 2\cdot\Omega(2^{n+1}) = \Omega(2^{(n+1)+1})$ and $|e_{n+1}|_{\mathtt{e}} = 4 + 2\cdot|e_n|_{\mathtt{e}} =_{i.h.} 4 + 2\cdot\Omega(2^{n+1}) = \Omega(2^{(n+1)+1})$. □

6 Conclusions

Contributions. For functional programming languages, it is generally assumed that the number of function calls, aka β-steps, is a reasonable cost model, since all other operations are dominated by the cost of β-steps. This paper shows that such a *negligible cost hypothesis* is less obvious than it seems at first sight, by considering constructors and pattern matching and showing that in CbN the number of pattern matching steps can be exponential in the number of β-steps. Furthermore, it shows that matching explosions are possible also in CbNeed, if evaluation is defined naively.

On the positive side, we showed that in CbV, and for a less naive formulation of CbNeed, the cost of pattern matching is indeed negligible: the number of matching steps is bilinear, that is, linear in the number of β-steps and in the size of the initial term. Summing up, we confirmed the negligible cost hypothesis for pattern matching, pointing out at the same time its subtleties. A novelty, is the use of cost models as a language design principle, to discriminate—in this paper—between otherwise equivalent formulations of CbNeed.

Coq and further extensions. The main motivation behind our work is the development of the analysis of the Coq abstract machine, that executes a language richer than the λ-calculus, including in particular pattern matching. To that aim, our CbNeed formalism, LINED, has to be further extended with fixpoints, and evaluation has to be generalized as to handle open terms and go under

abstraction. Here we omitted the study of fixpoints because they behave like β-redexes, being function calls, and their cost is not negligible, *i.e.* they have to be counted for complexity analyses. Moreover, all our results smoothly scale up to languages with fixpoints, without surprises. We plan to include them in a longer, journal version of this work. Open terms and evaluation under abstraction instead require more sophisticated machineries [10,9,5,11,12], whose adaptation to CbNeed and pattern matching is under development.

It would also be interesting to study other features of programming languages, such as first-class (delimited) continuations or other forms of effects, even if they are not part of the language executed by the Coq abstract machine.

Acknowledgements. This work has been partially funded by the ANR JCJC grant COCA HOLA (ANR-16-CE40-004-01).

References

1. Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. J. Funct. Program. 1(4), 375–416 (1991)
2. Accattoli, B.: An abstract factorization theorem for explicit substitutions. In: RTA. pp. 6–21 (2012)
3. Accattoli, B.: COCA HOLA. `https://sites.google.com/site/beniaminoaccattoli/coca-hola` (2016)
4. Accattoli, B.: The complexity of abstract machines. In: WPTE@FSCD 2016. pp. 1–15 (2016)
5. Accattoli, B.: The useful mam, a reasonable implementation of the strong λ-calculus. In: WoLLIC 2016. pp. 1–21. Springer (2016)
6. Accattoli, B., Barenbaum, P., Mazza, D.: Distilling Abstract Machines. In: ICFP 2014. pp. 363–376. ACM (2014)
7. Accattoli, B., Barras, B.: Environments and the Complexity of Abstract Machines. Accepted to PPDP 2017 (2017)
8. Accattoli, B., Bonelli, E., Kesner, D., Lombardi, C.: A Nonstandard Standardization Theorem. In: POPL. pp. 659–670 (2014)
9. Accattoli, B., Coen, C.S.: On the relative usefulness of fireballs. In: LICS 2015. pp. 141–155. IEEE Computer Society (2015)
10. Accattoli, B., Dal Lago, U.: (leftmost-outermost) beta reduction is invariant, indeed. Logical Methods in Computer Science 12(1) (2016)
11. Accattoli, B., Guerrieri, G.: Open call-by-value. In: APLAS 2016. pp. 206–226 (2016), `https://doi.org/10.1007/978-3-319-47958-3_12`
12. Accattoli, B., Guerrieri, G.: Implementing open call-by-value. Accepted at FSEN 2017 (2017)
13. Accattoli, B., Paolini, L.: Call-by-value solvability, revisited. In: FLOPS. pp. 4–16 (2012)
14. Accattoli, B., Sacerdoti Coen, C.: On the value of variables. In: WoLLIC 2014. pp. 36–50. Springer (2014)
15. Ariola, Z.M., Felleisen, M.: The call-by-need lambda calculus. J. Funct. Program. 7(3), 265–301 (1997)
16. Barras, B.: Auto-validation d'un système de preuves avec familles inductives. Ph.D. thesis, Université Paris 7 (1999)

17. Blelloch, G.E., Greiner, J.: Parallelism in sequential functional languages. In: FPCA 1995. pp. 226–237. ACM (1995)
18. Charguéraud, A., Pottier, F.: Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In: ITP 2015. pp. 137–153 (2015)
19. Cirstea, H., Kirchner, C.: The rewriting calculus - part I. Logic Journal of the IGPL 9(3), 339–375 (2001)
20. Coq Development Team: The coq proof-assistant reference manual, version 8.6 (2016), `http://coq.inria.fr`
21. Dal Lago, U., Martini, S.: Derivational complexity is an invariant cost model. In: FOPARA 2009. pp. 100–113 (2009)
22. Dal Lago, U., Martini, S.: On Constructor Rewrite Systems and the Lambda-Calculus. In: ICALP (2). pp. 163–174 (2009)
23. Danvy, O., Zerny, I.: A synthetic operational account of call-by-need evaluation. In: PPDP 2013. pp. 97–108. ACM (2013)
24. Fernández, M., Siafakas, N.: New developments in environment machines. Electr. Notes Theor. Comput. Sci. 237, 57–73 (2009)
25. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: ICFP 2002). pp. 235–246. ACM (2002)
26. Jay, C.B., Kesner, D.: First-class patterns. J. Funct. Program. 19(2), 191–225 (2009)
27. Jeannin, J., Kozen, D.: Computing with capsules. Journal of Automata, Languages and Combinatorics 17(2-4), 185–204 (2012)
28. Kesner, D.: The theory of calculi with explicit substitutions revisited. In: CSL. pp. 238–252 (2007)
29. Klop, J.W., van Oostrom, V., de Vrijer, R.C.: Lambda calculus with patterns. Theor. Comput. Sci. 398(1-3), 16–31 (2008)
30. Launchbury, J.: A natural semantics for lazy evaluation. In: POPL 1993. pp. 144–154. ACM Press (1993)
31. Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. J. Funct. Program. 8(3), 275–317 (1998)
32. Milner, R.: Local bigraphs and confluence: Two conjectures. Electr. Notes Theor. Comput. Sci. 175(3), 65–73 (2007)
33. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, MA, USA (2002)
34. Sands, D., Gustavsson, J., Moran, A.: Lambda calculi and linear speedups. In: The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. pp. 60–84. Springer (2002)
35. Sergey, I., Vytiniotis, D., Peyton Jones, S.L.: Modular, higher-order cardinality analysis in theory and practice. In: POPL '14. pp. 335–348 (2014)
36. Sestoft, P.: Deriving a lazy abstract machine. J. Funct. Program. 7(3), 231–264 (1997)
37. Wadsworth, C.P.: Semantics and pragmatics of the lambda-calculus. PhD Thesis, Oxford (1971), chapter 4
38. Walker, D.: In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages, chap. Substructural Type Systems, pp. 3–43. The MIT Press (2004)
39. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. 115(1), 38–94 (1994)