The Cost of Skeletal Call-By-Need, Smoothly

Beniamino Accattoli
Inria & LIX, École Polytechnique, Palaiseau, France

Francesco Magliocca
Università degli Studi di Napoli Federico II, Italy

Loïc Peyrot
IMDEA Software Institute, Madrid, Spain

Claudio Sacerdoti Coen
Alma Mater Studiorum – Università di Bologna, Italy

Abstract

Skeletal call-by-need is an optimization of call-by-need evaluation also known as "fully lazy sharing": when the duplication of a value has to take place, it is first split into "skeleton", which is then duplicated, and "flesh" which is instead kept shared.

Here, we provide two cost analyses of skeletal call-by-need. Firstly, we provide a family of terms showing that skeletal call-by-need can be asymptotically exponentially faster than call-by-need in both time *and* space; it is the first such evidence, to our knowledge.

Secondly, we prove that skeletal call-by-need can be implemented efficiently, that is, with bi-linear overhead. This result is obtained by providing a new smooth presentation of ideas by Shivers and Wand for the reconstruction of skeletons, which is then smoothly plugged into the study of an abstract machine following the distillation technique by Accattoli et al.

2012 ACM Subject Classification Theory of computation → Lambda calculus

Keywords and phrases λ-calculus, abstract machines, call-by-need, cost models

Digital Object Identifier 10.4230/LIPIcs.FSCD.2025.5

Related Version *Full Version*: `https://arxiv.org/abs/2505.09242` [18]

Supplementary Material *Software (Source Code)*: `https://github.com/Franciman/fullylazymad` archived at `swh:1:dir:50ff71d3e33d6febc5f53b8a6887aac6ef0be90f`

Funding *Claudio Sacerdoti Coen*: Supported by the INdAM-GNCS project "Modelli Composizionali per l'Analisi di Sistemi Reversibili Distribuiti" and by the Cost Action CA2011 EuroProofNet.

1 Introduction

Call-by-need evaluation of the λ-calculus was introduced by Wadsworth in 1971 as an optimization of the untyped λ-calculus [41]. It combines the advantages of both call-by-name and call-by-value: it avoids diverging on unused arguments, as in call-by-name, and when arguments are evaluated they do so only once, as in call-by-value. Such a combination is trickier to specify than call-by-name or call-by-value. In particular, Wadsworth's original presentation via graph reduction was not easy to manage.

In the 1990s, call-by-need was adopted by the then new Haskell language, and more manageable term-based calculi were developed by Launchbury [35] and Ariola et al. [21], and extended with non-determinism by Kutzner and Schmidt-Schauß [34]. Additionally, Sestoft studied abstract machines for call-by-need [38]. In 2014, Accattoli et al. introduced the *CbNeed linear substitution calculus* (CbNeed LSC) [4], a simplification of Ariola et al.'s calculus resting on the *at a distance* approach to explicit substitutions by Accattoli and Kesner [14]. It became a reference presentation, used in many recent studies [31, 24, 5, 33, 26,

13, 32, 25, 17, 16]. As far as this paper is concerned, the study in [4] is particularly relevant: beyond introducing the CbNeed LSC that we shall use, it also relates it with abstract machines for CbNeed, building a neat bisimulation between the two, deemed *distillation*.

Skeletal Call-by-Need. CbNeed was introduced together with a further optimization usually called *fully lazy sharing* and here rather referred to as *Skeletal CbNeed* (because *fully lazy sharing* is not really descriptive). The idea is to refine the duplication of a value v so as to duplicate only the *skeleton* of v while keeping its *flesh* shared, thus ending up sharing more than in ordinary call-by-need. As an example, if $v = \lambda x.\lambda y.zzx(yz)$ then the skeleton of v is $\texttt{skel}(v) := \lambda x.\lambda y.wx(yz)$ and its flesh is the context $\texttt{flesh}(v) := \mathsf{let}\ w = zz\ \mathsf{in}\ \langle\cdot\rangle$. That is, the flesh collects and removes the maximal (non-variable) sub-terms of v that are *free*, *i.e.* such that none of their free variables is captured in v.

Skeletal CbNeed is less studied than CbNeed because it is even trickier to define, as it requires to compute the skeletal decomposition of a value (that is, the split into skeleton and flesh). Early works by Turner [40], Hughes [30], and Peyton Jones [36] focus on implementative aspects. In the last ten years or so, various works provided operational insights. Balabonski developed an impressive theoretical analysis of Skeletal CbNeed, showing the equivalence of various presentations [22] and studying the relationship with Lévy's optimality [23]. Soon after that, Gundersen et al. showed that the *atomic* λ*-calculus*, a λ-calculus with sharing issued from deep inference technology, naturally specifies the duplication of the skeleton [29]. Next, Kesner et al. [32] merged the insights of the CbNeed LSC and the atomic λ-calculus obtaining a Skeletal CbNeed LSC, that is, a presentation at a distance of Skeletal CbNeed.

This Paper. The present work extends the operational research line of Balabonski, Gundersen et al., and Kesner et al. with cost analyses of Kesner et al.'s Skeletal CbNeed LSC, aiming at closing the gap with some of the earlier implementative studies at the same time. Generally speaking, we aim at adding Skeletal CbNeed to the list of concepts covered by the theory of cost-based analyses of abstract machines and sharing mechanisms by Accattoli and co-authors [4, 10, 19, 5, 7, 27, 12, 11, 8, 15]. We provide two analyses.

First Analysis: Exponential Time and Space Speed-Ups. The first analysis shows that in some cases Skeletal CbNeed is considerably more efficient than CbNeed. For that, we exhibit a family of λ-terms $\{t_n\}_{n\in\mathbb{N}}$ such that t_n evaluates in $\Omega(2^n)$ time and space in CbNeed, while requiring only $\mathcal{O}(n)$ space and time in Skeletal CbNeed. The literature only shows examples of single λ-terms where Skeletal CbNeed takes a few steps less than CbNeed, but never proves any asymptotic speed-up nor deals with space. Lastly, proving that our family evaluates within the given bounds requires a fine, non-trivial analysis of evaluation sequences.

Second Analysis: Bi-Linear Overhead. The second analysis shows that Skeletal CbNeed can be implemented efficiently, with an overhead that is *bi-linear*, *i.e.* linear in both the number of β-steps and the size of the initial term. This time, the same is true for CbNeed. The insight is that computing skeletons does require additional implementative efforts, but it comes at no asymptotic price. This work was indeed triggered by showing that the operational reconstruction of the skeleton at work in Gundersen et al. [29] and Kesner et al. [32] can be implemented in linear time. Their specifications rest on side conditions about variables in sub-terms that lead to non-linear overhead, if implemented literally. In fact, the linear time reconstruction of skeletons by Shivers and Wand [39], from 2010, avoids the side conditions and pre-dates the recent works. The insightful study in [39], however, is technical and not well-known. Here we simplify it, recasting it in the context of abstract machines.

Efficient Skeletal Decompositions, or Shivers and Wand Reloaded. Shivers and Wand present their ideas using graph-reduction for λ-terms. They reconstruct the skeletal decomposition via a visit of the value to skeletonize seen as a graph, based on two key points:

1. *Bi-directional edges*: all the edges of their graphs can be traversed in *both* directions. When term graphs are seen as mathematical objects, edges are often directed but the theoretical study can also look at edges in reverse. At the implementative level, however, things are different. For a parsimonious use of space, most graph-based implementations (*e.g.* the proof nets inspired ones in [5, 8, 20]) indeed restrict some edges to be traversed in one direction only (namely, parent to children), thus sparing some pointers. Therefore, bi-directional edges are an unusual approach, usually considered redundant.
2. *Abstractions, occurrences, and upward reconstruction*: in most graphical representations, there are edges between abstractions and the occurrences of their bound variables, usually directed only from the occurrences to the abstraction. Shivers and Wand's insight is that, with bi-directional edges, one can move from the abstraction back to the occurrences of the variable and then reconstruct the skeleton by visiting upward from the occurrences.

Shivers and Wand's study is graph-based and low-level, interleaved with code snippets. Here, we provide a new neat presentation of their algorithm as a simple high-level rewriting system over terms, circumventing graphs and low-level details. The reformulation of their study is the main technical innovation of this paper. While the key concepts are due to Shivers and Wand, we believe that our new presentation is a notable contribution in that it allows their concepts to blossom, simplifying their study and making them more widely accessible.

Distillation. We then smoothly lift the distillation-based study of CbNeed by Accattoli et al. [4] to Skeletal CbNeed, providing an abstract machines using the reloaded algorithm for skeletal decompositions as a black-box. The new abstract machine is then shown to be implementable within a bi-linear overhead, completing the second analysis.

The overall insight is that the apparently linear space inefficiency of adding pointers for bi-directional traversals of terms/graphs enables optimizations – namely, skeletal duplications – that can bring, in some cases, exponential speed-ups for both time and space.

Implementation. We provide an OCaml implementation of the abstract machine, and in particular of the reloaded skeleton reconstruction algorithm it rests upon. The implementation is there to validate the complexity analyses, as well as to integrate the low-level aspect of Shivers and Wand's study. The implementation is discussed in Appendix A.

Proofs. A long version with proofs in the Appendix is on arXiv [18].

2 Background: Call-by-Need

In this section, we recall CbNeed, presented as a strategy of the CbNeed linear substitution calculus (shortened to LSC). For the sake of simplicity, we do not distinguish here between the strategy and the calculus, and use *CbNeed LSC* to refer to both.

Accattoli and Kesner's LSC [1, 6] is a micro-step λ-calculus with explicit substitutions. *Micro-step* means that substitutions act on one variable occurrence at a time, rather than *small-step*, that is, on all occurrences at the same time. The CbNeed LSC was introduced by Accattoli et al. [4] as a simplified variant of Ariola et al.'s presentation of CbNeed [21]. The simplification is the use of rewriting rules *at a distance* – that is, adopting contexts in the root rules – that allow one to get rid of the commuting rewriting rules of Ariola et al.

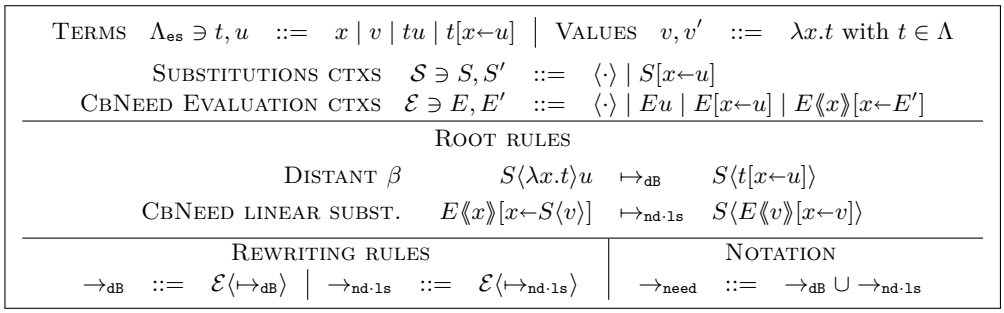

Figure 1 The Call-by-Need Linear Substitution Calculus (CbNeed LSC).

Terms and Contexts. The set of terms of the λ-calculus is noted Λ. The CbNeed LSC is defined in Fig. 1. The set of terms of the CbNeed LSC is noted Λ_{es}. They add *explicit substitutions* $t[x\leftarrow u]$ (shortened to ESs) to λ-terms, that is a more compact notation for $\mathsf{let}\ x = u\ \mathsf{in}\ t$, but where the order of evaluation between t and u is a priori not fixed; evaluation contexts shall fix it. The set $\mathtt{fv}(t)$ of *free* variables is defined as expected, in particular, $\mathtt{fv}(t[x\leftarrow u]) := (\mathtt{fv}(t) \setminus \{x\}) \cup \mathtt{fv}(u)$. Both $\lambda x.t$ and $t[x\leftarrow u]$ bind x in t, and terms are identified up to α-renaming. A term t is *closed* if $\mathtt{fv}(t) = \emptyset$, *open* otherwise. Meta-level capture-avoiding substitution is noted $t\{x\leftarrow u\}$. The size $|t|$ of t is the number of its constructs.

Note that values are only abstractions; this choice is standard in the literature on CbNeed. Note that, moreover, their bodies are λ-terms *without* ESs. This is in order to avoid dealing with calculating the skeleton of ESs in the next section. It is a harmless choice because evaluation (that creates ESs) never enters inside abstractions.

Contexts are terms with exactly one occurrence of the *hole* $\langle\cdot\rangle$, an additional constant, standing for a removed sub-term. We shall heavily use two notions of contexts: *substitution contexts* S, that are lists of ESs, and evaluation contexts E.

The main operation about contexts is *plugging* $E\langle t\rangle$ where the hole $\langle\cdot\rangle$ in context E is replaced by t. Plugging, as usual with contexts, can capture variables; for instance $((\langle\cdot\rangle t)[x\leftarrow u])\langle x\rangle = (xt)[x\leftarrow u]$. We write $E\langle\!\langle t\rangle\!\rangle$ when we want to stress that the context E does not capture the free variables of t.

Rewriting Rules. The reduction rules of the CbNeed LSC are slightly unusual as they use *contexts* both to allow one to reduce redexes located in sub-terms (via evaluation contexts E), which is standard, *and* to define the redexes themselves (via both substitution S and evaluation contexts E), which is less standard. This approach is called *at a distance* and related to cut-elimination on proof nets; see Accattoli [2, 3] for the link with proof nets. The notion of evaluation context E used for CbNeed is exactly the one by Ariola et al. [21], that extend call-by-name evaluation contexts with the production $E\langle\!\langle x\rangle\!\rangle[x\leftarrow E']$, which – by enterings ESs – is what enables the memoization aspect of CbNeed.

The *distant beta rule* $\mapsto_{dB}$ is essentially the β-rule, except that the argument goes into a new ES, rather then being immediately substituted, and that there can be a substitution context S in between the abstraction and the argument. Example: $(\lambda x.y)[y\leftarrow t]u \mapsto_{m} y[x\leftarrow u][y\leftarrow t]$. One with on-the-fly α-renaming is $(\lambda x.y)[y\leftarrow t]y \mapsto_{m} z[x\leftarrow y][z\leftarrow t]$.

The *linear substitution rule by need* $\mapsto_{nd\cdot ls}$ replaces a single variable occurrence by a copy of the value in the ES, commuting the substitution context around the value (if any) out of the ES. Example: $(xx)[x\leftarrow \mathtt{I}[y\leftarrow t]] \mapsto_{nd\cdot ls} (\mathtt{I}x)[x\leftarrow \mathtt{I}][y\leftarrow t]$.

The use of substitution contexts S in the root rules is what allows one to avoid the two commuting rules of Ariola et al. [21], namely (rephrasing their let-expressions as ESs) $t[x\leftarrow s]u \mapsto_{let\text{-}C} (tu)[x\leftarrow s]$ and $t[y\leftarrow u[x\leftarrow s]] \mapsto_{let\text{-}A} t[y\leftarrow u][x\leftarrow s]$.

The two root rules $\mapsto_{\mathtt{dB}}$ and $\mapsto_{\mathtt{nd{\cdot}ls}}$ are then closed by evaluation contexts E. In Fig. 1, we use the compact notation $\to_{\mathtt{dB}} := \mathcal{E}\langle\mapsto_{\mathtt{dB}}\rangle$ to denote that $\to_{\mathtt{dB}}$ is defined as $E\langle t\rangle \to_{\mathtt{dB}} E\langle u\rangle$ if $t \mapsto_{\mathtt{dB}} u$, and similarly for the other rule.

3 Skeletal Call-by-Need

In this section, we present our version of Skeletal CbNeed, which is a very minor variant of the one by Kesner et al. [32] (the difference is discussed at the end of the section), itself defined by tweaking the CbNeed LSC via the notion of skeleton.

The basic idea of skeletal CbNeed is that every value $v := \lambda x.t$ can be split in two: the skeleton $\lambda x.u$, which is a sort of sub-term of t, and the flesh S, which is a substitution context collecting the maximal sub-terms of v that do not depend on x. The flesh is extracted from v before duplicating it, as to avoid duplicating the code (and possibly redexes) of the flesh, thus increasing sharing. Some definitions are in order.

Free Sub-Terms. Extractable sub-terms are called *free sub-terms*, defined next. We actually need a parametrized notion of free sub-term, which is relative to a set of variables $\mathtt{V}$ that are supposed not to occur in the sub-term.

▶ **Definition 1** (Maximally free sub-terms). *Let $t \in \Lambda$ and $\mathtt{V}$ be a set of variables. A sub-term u of t is (the term part of) a decomposition $t = C\langle u\rangle$ for some context C; additionally, u is* $\mathtt{V}$-free in t *if no variable in $\mathtt{fv}(u)$ is captured by C and if $\mathtt{fv}(u) \cap \mathtt{V} = \emptyset$, and* maximally $\mathtt{V}$-free *if it is not a strict sub-term of any other $\mathtt{V}$-free sub-term in t.*

Skeleton. The skeletal decomposition of a term t is the splitting of t into its maximal free sub-terms on one side, collected as a substitution context S called *the flesh*, and what is left of t after the removal, that is, its *skeleton*. As for free sub-terms, we rather define parametrized notions of skeletal decomposition and skeleton. Then, we specialize the definitions for values.

▶ **Definition 2** ((Relative) Skeleton). *Let $\mathtt{V}$ be a set of variables. The skeletal decomposition of t relative to a set of variables $\mathtt{V}$ is the pair (u, S) of an ordinary λ-term u and a substitution context S defined as $\mathtt{skdec}^{\mathtt{V}}(t) := (x, \langle\cdot\rangle[x\leftarrow t])$ with x fresh, if $\mathtt{fv}(t) \cap \mathtt{V} = \emptyset$ and t is not a variable, otherwise:*

$$\begin{array}{rcll} \mathtt{skdec}^{\mathtt{V}}(x) & := & (x, \langle\cdot\rangle) & \\ \mathtt{skdec}^{\mathtt{V}}(\lambda x.u) & := & (\lambda x.s, S) & \text{where } \mathtt{skdec}^{\mathtt{V}\cup\{x\}}(u) = (s, S) \\ \mathtt{skdec}^{\mathtt{V}}(us) & := & (rp, S'\langle S\rangle) & \text{where } \mathtt{skdec}^{\mathtt{V}\cup\{x\}}(u) = (r, S) \text{ and } \mathtt{skdec}^{\mathtt{V}\cup\{x\}}(s) = (p, S') \end{array}$$

The skeleton, the flesh, and the skeletal decomposition of $\lambda x.t$ are defined respectively as $\mathtt{skel}(\lambda x.t) := \lambda x.u$, $\mathtt{flesh}(\lambda x.t) := S$, and $\mathtt{skdec}(\lambda x.t) := (\mathtt{skel}(\lambda x.t), \mathtt{flesh}(\lambda x.t))$, where $\mathtt{skdec}^{\{x\}}(t) = (u, S)$.

Skeletal CbNeed. We adopt the presentation of skeletal CbNeed by Kesner et al. [32] (refered to as *fully lazy CbNeed* by them), that adds a new *skeletal (explicit) substitution* $t\langle x\leftarrow v\rangle$ containing *skeletal values*, that is, values that are equal to their skeleton. The language of terms and the strategy is defined in Fig. 2. The new set of terms is noted $\Lambda_{\mathtt{sk}}$.

Terms	$\Lambda_{\mathtt{sk}} \ni t, u$	$::=$	$x \mid v \mid tu \mid t[x{\leftarrow}u] \mid t\langle x{\leftarrow}v\rangle$ with $v = \mathtt{skel}(v)$
Values	v, v'	$::=$	$\lambda x.t$ with $t \in \Lambda$
Substitutions ctxs	$\mathcal{S}_{\mathtt{sk}} \ni S, S'$	$::=$	$\langle\cdot\rangle \mid S[x{\leftarrow}u] \mid S\langle x{\leftarrow}v\rangle$
Skeletal eval. ctxs	$\mathcal{E}_{\mathtt{sk}} \ni E, E'$	$::=$	$\langle\cdot\rangle \mid Eu \mid E[x{\leftarrow}u] \mid E\langle x{\leftarrow}v\rangle \mid E\langle\!\langle x\rangle\!\rangle[x{\leftarrow}E']$

Root rules

Distant β	$S\langle\lambda x.t\rangle u$	$\mapsto_{\mathtt{dB}}$	$S\langle t[x{\leftarrow}u]\rangle$
Skeletonization	$E\langle\!\langle x\rangle\!\rangle[x{\leftarrow}S\langle v\rangle]$	$\mapsto_{\mathtt{sk}}$	$S\langle S'\langle E\langle\!\langle x\rangle\!\rangle\langle x{\leftarrow}v'\rangle\rangle\rangle$ with $\mathtt{skdec}(v) = (v', S')$
Skeletal subst.	$E\langle\!\langle x\rangle\!\rangle\langle x{\leftarrow}v\rangle$	$\mapsto_{\mathtt{ss}}$	$E\langle\!\langle v\rangle\!\rangle\langle x{\leftarrow}v\rangle$

Rewriting rules

$\to_{\mathtt{sk{\cdot}dB}}$	$::=$	$\mathcal{E}_{\mathtt{sk}}\langle\mapsto_{\mathtt{dB}}\rangle$	$\to_{\mathtt{sk}}$	$::=$	$\mathcal{E}_{\mathtt{sk}}\langle\mapsto_{\mathtt{sk}}\rangle$
$\to_{\mathtt{ss}}$	$::=$	$\mathcal{E}_{\mathtt{sk}}\langle\mapsto_{\mathtt{ss}}\rangle$			

Notations

$\to_{\mathtt{sk{\cdot}need}} ::= \to_{\mathtt{sk{\cdot}dB}} \cup \to_{\mathtt{sk}} \cup \to_{\mathtt{ss}}$

$[\![x{\leftarrow}t]\!]$ stands for either $[x{\leftarrow}t]$ or $\langle x{\leftarrow}v\rangle$

Figure 2 The skeletal CbNeed strategy $\to_{\mathtt{sk{\cdot}need}}$.

Root structural equalities

Expl. subst.	$E\langle t[x{\leftarrow}s]\rangle$	$\equiv_1$	$E\langle t\rangle[x{\leftarrow}s]$	if $\mathtt{dom}(E) \cap \mathtt{fv}(t[x{\leftarrow}s]) = \emptyset$ and $x \notin \mathtt{fv}(E)$
Skel. subst.	$E\langle t\langle x{\leftarrow}v\rangle\rangle$	$\equiv_2$	$E\langle t\rangle\langle x{\leftarrow}v\rangle$	if $\mathtt{dom}(E) \cap \mathtt{fv}(t\langle x{\leftarrow}v\rangle) = \emptyset$ and $x \notin \mathtt{fv}(E)$

Closure rules

$$\frac{t \equiv_1 u}{t \equiv u} \qquad \frac{t \equiv_2 u}{t \equiv u} \qquad \frac{}{t \equiv t}\ \text{ref} \qquad \frac{t \equiv u}{u \equiv t}\ \text{sym} \qquad \frac{t \equiv u \qquad u \equiv s}{t \equiv s}\ \text{tr} \qquad \frac{t \equiv u}{E\langle t\rangle \equiv E\langle u\rangle}\ \text{ev}$$

Figure 3 Structural equivalence $\equiv$.

Notation: when we need to treat explicit and skeletal substitution together, we use the notation $[\![x{\leftarrow}t]\!]$, which stands for either $[x{\leftarrow}t]$ or (with a slight abuse) $\langle x{\leftarrow}v\rangle$.

The CbNeed substitution rule $\to_{\mathtt{nd{\cdot}ls}}$ for a redex $E\langle\!\langle x\rangle\!\rangle[x{\leftarrow}S\langle v\rangle]$ is refined via two rules. The first *skeletonization* rule $\to_{\mathtt{sk}}$ does three further mini tasks at once: firstly, it decomposes v in its skeleton v' and its flesh S' (thus obtaining $E\langle\!\langle x\rangle\!\rangle[x{\leftarrow}S\langle S'\langle v\rangle\rangle]$); secondly, it re-organizes the term commuting S and S' out of the ES (obtaining $S\langle S'\langle E\langle\!\langle x\rangle\!\rangle[x{\leftarrow}v']\rangle\rangle$); thirdly, it turns the ES into a skeletal substitutions, finally producing $S\langle S'\langle E\langle\!\langle x\rangle\!\rangle\langle x{\leftarrow}v'\rangle\rangle\rangle$.

The second *skeletal substitution* rule $\to_{\mathtt{ss}}$ simply replaces x with v'.

Note that evaluation contexts E are adapted to include skeletal substitutions, but that they do not enter inside them, since their content is always a value. The distant β rule for Skeletal CbNeed is noted $\to_{\mathtt{sk{\cdot}dB}}$ to distinguish it from the one for CbNeed. The Skeletal CbNeed reduction $\to_{\mathtt{sk{\cdot}need}}$ is the union of $\to_{\mathtt{sk{\cdot}dB}}$, $\to_{\mathtt{sk}}$, and $\to_{\mathtt{ss}}$.

The next proposition is an easy adaptation of the one for CbNeed in Accattoli et al. [4].

▶ **Proposition 3.** *The reduction $\to_{\mathtt{sk{\cdot}need}}$ is deterministic.*

Structural Equivalence. To study the relationship with abstract machines, we shall need the concept of structural equivalence, defined in Fig. 3. The concept is standard for λ-calculi with ESs at a distance, and it also standard to use it in relationship with abstract machines, as systematically done by Accattoli et al. [4]. Intuitively, structural equivalence allows one to move explicit/skeletal substitutions around because the move does not change the behaviour of the term. This fact is captured by the following strong bisimulation property, proved by an immediate adaptation of the similar property in [4].

▶ **Proposition 4** ($\equiv$ is a strong bisimulation). *Let $a \in \{\mathtt{dB}, \mathtt{sk}, \mathtt{ss}\}$. If $t \equiv u$ and $t \to_a t'$ then there exists u' such that $u \to_a u'$ and $t' \equiv u'$.*

Difference with Kesner et al's Skeletal CbNeed [32]. There is a small difference between our definition of Skeletal CbNeed and Kesner et al.'s, namely in the definition of skeletons. For us, the skeletal decomposition of, say, $v := \lambda x.yx(zz)$ is $(\lambda x.yxw, \langle\cdot\rangle[w\leftarrow zz])$, while for them it is $(\lambda x.y'xw, \langle\cdot\rangle[y'\leftarrow y][w\leftarrow zz])$, that is, they *flesh-out* also single variable occurrences while we do *not*, obtaining a slightly more parsimonious decomposition.

This difference, however, has essentially no impact on the theory, nor on the asymptotic costs. In particular, the operational equivalence of Skeletal CbNeed with respect to CbNeed and CbN is preserved. The proof of Kesner et al., indeed, works for all decompositions such that substituting the flesh into the skeleton recovers the original value; our decomposition has this property. Therefore, we can import the following result.

▶ **Theorem 5** (Operational equivalence [32]). *Two λ-terms t and u are CbN observational equivalent if and only if they are Skeletal CbNeed observational equivalent.*

4 Skeletal CbNeed Can Bring an Exponential Asymptotic Speed-Up

In this section, we show that Skeletal CbNeed can be both exponentially faster and use exponentially less space than CbNeed, taking as reference cost models (following the terminology of Accattoli et al. [15]):

- *Abstract time*, that is the number of (distant) β-steps;
- *Ink space*, that is, the maximum size of the involved terms.

By smoothly adapting results in the literature [37, 9, 28], these can be proved to be reasonable cost models for time for CbNeed and for (super-)linear space for both CbNeed and Skeletal CbNeed. In fact, the relevant part of proving that abstract time is reasonable for Skeletal CbNeed shall be our own Theorem 31 at the end of the paper.

We prove the result by defining a family of terms $\{t_n\}_{n\in\mathbb{N}}$ and showing that t_n evaluates in an exponential number of $\mathtt{dB}$ steps and producing terms of exponential size in CbNeed, while it uses only a linear number of $\mathtt{sk}\cdot\mathtt{dB}$ steps and terms of linear size in Skeletal CbNeed.

The Family. We shall use the abbreviations $\mathtt{I} := \lambda x.x$ and $\gamma := \lambda y.\lambda z.y\mathtt{I}(y\mathtt{I})z$. Next, we define an auxiliary family $\{u_n\}_{n\in\mathbb{N}}$ as follows: $u_0 := \mathtt{I}$ and $u_{n+1} := \gamma u_n$. Lastly, the actual family is defined as $t_n := (\lambda x.x\mathtt{I}(x\mathtt{I}))u_n$.

Our result is that the term t_n evaluates in $\mathcal{O}(n)$ $\to_{\mathtt{sk}\cdot\mathtt{dB}}$-steps (namely $6n+4$ steps) with skeletal CbNeed and $\Omega(2^n)$ $\to_{\mathtt{dB}}$-steps (namely $8\times 2^n + n - 4$ steps) with CbNeed. This is shown via quite technical statements. Firstly, note that $t_n \to_{\mathtt{dB}} (x\mathtt{I}(x\mathtt{I}))[x\leftarrow u_n]$; it is actually for $(x\mathtt{I}(x\mathtt{I}))[x\leftarrow u_n]$ that we shall prove bounds. These bounds are expressed stating that there are families of evaluation contexts $S^{\mathtt{nd}}_n$ and $S^{\mathtt{sk}}_n$ (one for CbNeed and one for Skeletal CbNeed) such that $(x\mathtt{I}(x\mathtt{I}))[x\leftarrow u_n]$ reduces to $S^{\mathtt{nd}}_n\langle\mathtt{I}\rangle$ and $S^{\mathtt{sk}}_n\langle\mathtt{I}\rangle$ (which is a normal form in both cases) in $6n+3$ and $8\times 2^n + n - 3$ steps respectively. The results about space are obtained by showing that $S^{\mathtt{nd}}_n$ has size exponential in n, while $S^{\mathtt{sk}}_n$ is linear in n.

Evaluation in Skeletal CbNeed. The skeletal case is, perhaps surprisingly, the easiest case for which one can find an inductive invariant leading to the bound.

▶ **Lemma 6.** *Let $n\in\mathbb{N}$ and $s_n := (x\mathtt{I}(x\mathtt{I}))[x\leftarrow u_n]$. Then there is a substitution context $S^{\mathtt{sk}}_n$ and an evaluation $\mathtt{e}_n : s_n \to^*_{\mathtt{sk}\cdot\mathtt{need}} S^{\mathtt{sk}}_n\langle\mathtt{I}\rangle$ such that $|\mathtt{e}_n|_{\mathtt{sk}\cdot\mathtt{dB}} = 6n+3$ and $|S^{\mathtt{sk}}_n| = \mathcal{O}(n)$.*

Marked terms	$\Lambda_\circ \ni m, n$	$::=$	$x \mid x^\circ \mid \lambda x.m \mid \lambda^\circ x.m \mid mn \mid m \circ n \mid \overline{m}^{\Uparrow}$
Marked contexts	$\mathcal{M} \ni M$	$::=$	$\langle\cdot\rangle \mid \lambda x.M \mid \lambda^\circ x.M \mid Mn \mid mM$ $\mid M \circ n \mid m \circ M \mid \overline{M}^{\Uparrow}$

Figure 4 Marked terms and contexts.

Evaluation in CbNeed. For the non-skeletal case, the bound is formulated via two families of substitution contexts. For $n \in \mathbb{N}$, the first family is given by $S_0 := \langle\cdot\rangle[x_0 \leftarrow \mathtt{I}]$ and $S_{n+1} := S_n\langle\langle\cdot\rangle[x_{n+1} \leftarrow s_{n+1}]\rangle$ with $s_{n+1} := \lambda y_n.x_n\mathtt{I}(x_n\mathtt{I})y_n$. The second family $S_n^{\mathtt{nd}}$ is defined in the proof of the lemma, itself relying on an auxiliary lemma in the technical report [18]. The statement is parametrized by an evaluation context E for the induction to go through, but the relevant case is the one with E empty.

▶ **Lemma 7.** *Let $n \in \mathbb{N}$ and E be an evaluation context. Then there exist a substitution context $S_n^{\mathtt{nd}}$ and an evaluation $\mathtt{e}_n : E\langle\!\langle x_n\mathtt{I}(x_n\mathtt{I})\rangle\!\rangle[x_n \leftarrow u_n] \to^*_{\mathtt{need}} S_n\langle E\langle\!\langle S_n^{\mathtt{nd}}\langle\mathtt{I}\rangle\rangle\!\rangle\rangle$ such that $|\mathtt{e}_n|_{\mathtt{dB}} = 8 \times 2^n + n - 5$ and $|S_n^{\mathtt{nd}}| = \Omega(2^n)$.*

The next statement sums up the results, and is based on the result at the end of the paper stating that Skeletal CbNeed can be implemented in bilinear time, that is, linear in the size $|t_n|$ of the initial term and in the number of $\mathtt{sk{\cdot}dB}$ steps (Theorem 31).

▶ **Theorem 8** (Instance of exponential skeletal speed-up for both time and space)**.** *There is a family $\{t_n\}_{n\in\mathbb{N}}$ of λ-terms such that t_n normalizes in $\mathcal{O}(n)$ abstract time and $\mathcal{O}(n)$ ink space with Skeletal CbNeed and $\Omega(2^n)$ abstract time and $\Omega(2^n)$ ink space with CbNeed.*

Proof.

- For CbNeed, by Lemma 7 there is an evaluation sequence:

  $$t_n = (\lambda x.x\mathtt{I}(x\mathtt{I}))u_n \to_{\mathtt{dB}} (x\mathtt{I}(x\mathtt{I}))[x \leftarrow u_n] \to^*_{\mathtt{need}} S_n\langle S_n^{\mathtt{nd}}\langle\mathtt{I}\rangle\rangle$$

  taking $k_n := 8 \times 2^n + n - 4 = \Omega(2^n)$ $\mathtt{dB}$-steps and such that $|S_n^{\mathtt{nd}}| = \Omega(2^n)$.
- For Skeletal CbNeed, by Lemma 6 there is an evaluation sequence:

  $$t_n = (\lambda x.x\mathtt{I}(x\mathtt{I}))u_n \to_{\mathtt{sk{\cdot}dB}} (x\mathtt{I}(x\mathtt{I}))[x \leftarrow u_n] \to^*_{\mathtt{sk{\cdot}need}} S_n^{\mathtt{sk}}\langle\mathtt{I}\rangle$$

  taking $k_n := 6n + 4 = \mathcal{O}(n)$ $\mathtt{sk{\cdot}dB}$-steps and such that $|S_n^{\mathtt{sk}}| = \mathcal{O}(n)$. Note that the calculus has no garbage collection rule, so the size of terms diminishes only by $\mathtt{sk{\cdot}dB}$ steps, and of a constant amount (a $\mathtt{sk{\cdot}dB}$ step removes two constructs and adds one). Therefore, the maximum size of terms is bounded by the size of the final term plus the number of steps. That is, the space cost is $\mathcal{O}(n)$. ◀

5 A Rewriting-Based Algorithm for Computing the Skeleton

In this section, we first define *marked* terms and a marked reformulation of skeletal decompositions. Then, we use the marked setting to give a graphically-inspired rewriting system computing the skeletal decomposition of a value, giving a neat new presentation of ideas first developed by Shivers and Wand [39]. The rewriting system formalizes an algorithm. It is presented as a rewriting system as to smoothly manage it as an operational semantics notion.

Marked Skeleton. Marked terms and contexts are defined in Fig. 4. The idea is to incrementally mark with a tag $\circ$ the constructs belonging to the skeleton during a visit of the term which is allowed to jump from an abstraction to the occurrences of the variables, since these are usually connected in a graphical representation.

Constructs of the λ calculus are then either the usual ones (referred to as *unmarked*) or marked with $\circ$, which means that the constructor is part of the skeleton. Additionally, there is an extra *up* construct $\overline{t}^{\Uparrow}$ that shall be used to indicate the frontier of the visit of the term.

In the marked setting, we can recast skeletal decompositions by marking as white the constructs belonging to the skeleton and leaving unmarked all the constructs of the flesh.

▶ **Definition 9** (Marked skeleton). *Let* $\mathtt{V}$ *be a set of variables. The marked skeleton of* t *relative to* $\mathtt{V}$ *is defined as* $\mathtt{mskel}^{\mathtt{V}}(t) := t$ *if* $\mathtt{fv}(t) \cap \mathtt{V} = \emptyset$*, otherwise:*

$$\begin{array}{rcl|rcl} \mathtt{mskel}^{\mathtt{V}}(x) & := & x^{\circ} & \mathtt{mskel}^{\mathtt{V}}(tu) & := & \mathtt{mskel}^{\mathtt{V}}(m) \circ \mathtt{mskel}^{\mathtt{V}}(n) \\ \mathtt{mskel}^{\mathtt{V}}(\lambda x.t) & := & \lambda^{\circ} x.\mathtt{mskel}^{\mathtt{V}\cup\{x\}}(t) & & & \end{array}$$

The marked skeleton of $\lambda x.t$ *is defined as* $\mathtt{mskel}(\lambda x.t) := \lambda^{\circ} x.\mathtt{mskel}^{\{x\}}(t)$.

Example: for $v := \lambda x.\lambda y.zzx(yz)$, one has $\mathtt{mskel}(v) = \lambda^{\circ} x.\lambda^{\circ} y.(zz) \circ x^{\circ} \circ (y^{\circ} \circ z)$.

From the Marked Skeleton to the Skeletal Decomposition. Next, we turn marked skeletons into skeletal decompositions, by extracting unmarked sub-terms and turning them into the flesh, while removing the marks from the skeleton. We shall do this via a *splitting function*, defined below, which operates on (parametrized) marked skeletons. For that, we give a lemma guaranteeing that the unmarked sub-terms of marked skeletons can be recognized from their top constructor, that is in $\mathcal{O}(1)$, thus removing the need to inspect them.

▶ **Lemma 10** (Being unmarked is downward closed in marked skeletons). *Let* $\mathtt{mskel}^{\mathtt{V}}(t) = M\langle m \rangle$ *with* m *starting unmarked. Then* m *has no marks.*

The lemma implies the correctness of the first clause of the following definition, as it ensures that in that case m has no marks.

▶ **Definition 11** (Splitting). *Let* $m = \mathtt{mskel}^{\mathtt{V}}(t)$ *for some* t *and* $\mathtt{V}$*. The following splitting function separates the skeleton from the flesh and removes all marks:*

$$\begin{array}{rcll} \mathtt{split}(m) & := & (z, \langle\cdot\rangle[z \leftarrow m]) & \text{if } m \text{ starts unmarked, is not a variable, and } z \text{ is fresh;} \\ \mathtt{split}(x) & := & (x, \langle\cdot\rangle); & \\ \mathtt{split}(x^{\circ}) & := & (x, \langle\cdot\rangle); & \\ \mathtt{split}(\lambda^{\circ} x.n) & := & (\lambda x.u, S) & \text{with } \mathtt{split}(n) = (u, S); \\ \mathtt{split}(m_1 \circ m_2) & := & (t_1 t_2, S'\langle S\rangle) & \text{with } \mathtt{split}(m_1) = (t_1, S) \text{ and } \mathtt{split}(m_2) = (t_2, S'). \end{array}$$

Example: $\mathtt{split}(\lambda^{\circ} x.\lambda^{\circ} y.(zz) \circ x^{\circ} \circ (y^{\circ} \circ z)) = (\lambda x.\lambda y.wx(yz), \langle\cdot\rangle[w \leftarrow zz])$.

We can now prove that the marked notions allow one to retrieve the standard ones.

▶ **Proposition 12** (Soundness of the marked skeleton). *Let* $t \in \Lambda$ *and* $\mathtt{V}$ *be a set of variables.*

1. Auxiliary parametrized statement*:* $\mathtt{split}(\mathtt{mskel}^{\mathtt{V}}(t)) = \mathtt{skdec}^{\mathtt{V}}(t)$*;*
2. Soundness*:* $\mathtt{skdec}(\lambda x.t) = \mathtt{split}(\mathtt{mskel}(\lambda x.t))$.

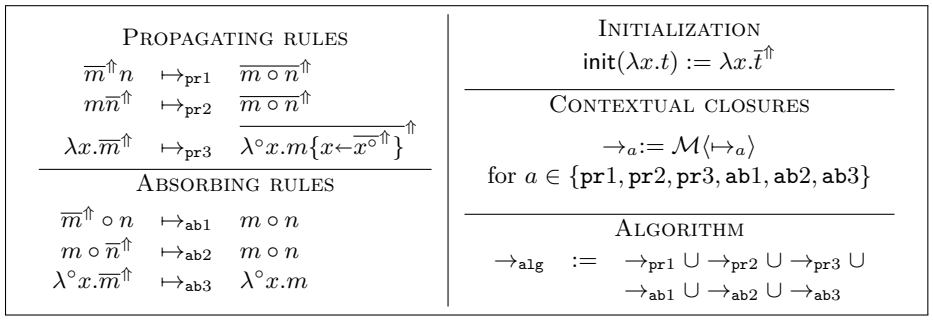

Figure 5 Rewriting rules for marked terms.

Computing the Marked Skeleton. We shall now define a parallel algorithm for computing the marked skeleton via a rewriting system on marked terms. The definition is in Fig. 5. The algorithm shall be applied to abstractions, say to $\lambda x.t$. The algorithm is described by inserting the *up* symbols $\Uparrow$, denoting where the algorithm is operating, and propagating through the term. At the beginning, there is exactly one $\Uparrow$, as the algorithm starts on $\lambda x.\overline{t}^{\Uparrow}$.

The algorithm has two sets of rules. Propagation rules turn the encountered unmarked constructs into marked ones. The interesting rule is $\to_{\mathtt{pr3}}$ for abstractions, that is also the very first one to be applied on the initial term $\lambda x.\overline{t}^{\Uparrow}$. Its effect is that λx is marked as $\lambda^{\circ} x$ and all the occurrences of x in t are also marked, obtaining $\lambda^{\circ} x.t\{x \leftarrow x^{\circ}\}$; this is how we recast the role played by Shivers and Wand's bi-directional edges connecting abstractions and variables. Essentially, propagation rules move $\Uparrow$ upwards but they also add $\Uparrow$ at the bottom of the syntax tree on the variable occurrences associated to marked abstractions.

Absorption rules, instead, remove the $\Uparrow$ symbol when it encounters a marked construct, as it means that the concerned thread of the algorithm is passing where another thread already passed before, thus the concerned thread becomes redundant and can be terminated.

Intuitively, the set of $\Uparrow$-sub-terms is the frontier of where the parallel algorithm is operating. When the algorithm is over, no internal node is marked with $\Uparrow$.

The algorithm is parallel in that the initial substitution $t\{x \leftarrow x^{\circ}\}$ and rule $\mapsto_{\mathtt{pr3}}$ might introduce many occurrences of $\Uparrow$, which can be propagated independently.

As an example, consider $v := \lambda x.\lambda y.zzx(yz)$, for which one has, for instance:

$$
\begin{array}{rlll}
\mathsf{init}(v) & = & \lambda x.\overline{\lambda y.zzx(yz)}^{\Uparrow} & \to_{\mathtt{pr3}} \overline{\lambda^{\circ} x.\lambda y.zz\overline{x^{\circ}}^{\Uparrow}(yz)}^{\Uparrow} \\
& \to_{\mathtt{pr2}} & \overline{\lambda^{\circ} x.\lambda y.\overline{(zz) \circ x^{\circ}}^{\Uparrow}(yz)}^{\Uparrow} & \to_{\mathtt{pr1}} \overline{\lambda^{\circ} x.\lambda y.\overline{(zz) \circ x^{\circ} \circ (yz)}^{\Uparrow}}^{\Uparrow} \\
& \to_{\mathtt{pr3}} & \overline{\lambda^{\circ} x.\overline{\lambda^{\circ} y.(zz) \circ x^{\circ} \circ (\overline{y^{\circ}}^{\Uparrow} z)}^{\Uparrow}}^{\Uparrow} & \to_{\mathtt{pr1}} \overline{\lambda^{\circ} x.\overline{\lambda^{\circ} y.(zz) \circ x^{\circ} \circ \overline{y^{\circ} \circ z}^{\Uparrow}}^{\Uparrow}}^{\Uparrow} \\
& \to_{\mathtt{ab2}} \to_{\mathtt{ab3}} & \overline{\lambda^{\circ} x.\overline{\lambda^{\circ} y.(zz) \circ x^{\circ} \circ (y^{\circ} \circ z)}^{\Uparrow}}^{\Uparrow} & = \overline{\mathtt{mskel}(v)}^{\Uparrow}
\end{array}
$$

▶ **Proposition 13.** *Reduction $\to_{\mathtt{alg}}$ is strongly normalizing and diamond (thus confluent).*

Proof.

1. *Strong normalization.* Let m be a marked term. As terminating measure, consider $(|m|_{\neg\circ}, |m|_{\Uparrow})$ where $|m|_{\neg\circ}$ and $|m|_{\Uparrow}$ are the number of unmarked and $\Uparrow$ constructs in m. Note that the propagation rules decrease $|m|_{\neg\circ}$ while the absorption rules decrease $|m|_{\Uparrow}$ and leave $|m|_{\neg\circ}$ unchanged. Then $\to_{\mathtt{alg}}$ is strongly normalizing.
2. *Diamond.* We prove the diamond property, which implies confluence. Clearly, redexes cannot duplicate or erase each other. There are only two critical pairs, namely:

$$\begin{array}{ccc} M\langle\overline{m}^{\Uparrow}\overline{n}^{\Uparrow}\rangle & \xrightarrow{\texttt{pr1}} & M\langle m\circ\overline{n}^{\Uparrow}\rangle \\ {\scriptstyle\texttt{pr2}}\downarrow & & \vdots{\scriptstyle\texttt{ab2}} \\ M\langle\overline{m}^{\Uparrow}\circ n\rangle & \xrightarrow{\texttt{ab1}} & M\langle m\circ n\rangle \end{array} \qquad \begin{array}{ccc} M\langle\overline{m}^{\Uparrow}\circ\overline{n}^{\Uparrow}\rangle & \xrightarrow{\texttt{ab1}} & M\langle m\circ\overline{n}^{\Uparrow}\rangle \\ {\scriptstyle\texttt{ab2}}\downarrow & & \vdots{\scriptstyle\texttt{ab2}} \\ M\langle\overline{m}^{\Uparrow}\circ n\rangle & \xrightarrow{\texttt{ab1}} & M\langle m\circ n\rangle \end{array}$$

◀

To state the correctness of the algorithm and give a bound on the number of its steps, we need two definitions.

▶ **Definition 14.** *Let m be a marked term.*

- Starts with $\circ$/unmarked/$\Uparrow$: *m starts with $\circ$ (resp.* unmarked, *resp.* with $\Uparrow$*) if it has shape x°, $\lambda^{\circ}x.n$, or $n\circ o$ (resp. x, $\lambda x.n$, or no, resp. $\overline{m}^{\Uparrow}$).*
- White size*: the white size $|m|_{\circ}$ of m is the number of $\circ$ constructs in m.*

The following lemma is the key step in the proof of correctness of the algorithm. It is a technical lemma, the statement of which is more easily understood by looking at the proof of the following theorem.

▶ **Lemma 15** (Crucial). *Let t be a term, $\mathtt{V}$ a set of variables, $x\in\mathtt{fv}(t)\setminus\mathtt{V}$, $m:=\mathtt{mskel}^{\mathtt{V}}(t)$, and $n:=\mathtt{mskel}^{\mathtt{V}\cup\{x\}}(t)$. Then:*

$$m\{x\leftarrow\overline{x^{\circ}}^{\Uparrow}\}\to_{\mathtt{alg}}^{k}\begin{cases}\overline{n}^{\Uparrow} & \text{with } k=|n|_{\circ}-|m|_{\circ}-1\text{, if } m \text{ starts unmarked;}\\ n & \text{with } k=|n|_{\circ}-|m|_{\circ}\text{, if } m \text{ starts with } \circ.\end{cases}$$

▶ **Theorem 16** (Soundness of the algorithm). *Let t be a term. Then* $\mathsf{init}(\lambda x.t)\to_{\mathtt{alg}}^{n}\overline{\mathtt{mskel}(\lambda x.t)}^{\Uparrow}$ *with $n=|\mathtt{mskel}(\lambda x.t)|_{\circ}$.*

Proof. By definition $\mathsf{init}(\lambda x.t)=\lambda x.\overline{t}^{\Uparrow}\to_{\mathtt{pr3}}\overline{\lambda^{\circ}x.t\{x\leftarrow\overline{x^{\circ}}^{\Uparrow}\}}^{\Uparrow}=\overline{\lambda^{\circ}x.\mathtt{mskel}^{\emptyset}(t)\{x\leftarrow\overline{x^{\circ}}^{\Uparrow}\}}^{\Uparrow}$, while $\mathtt{mskel}(\lambda x.t)=\lambda^{\circ}x.\mathtt{mskel}^{\{x\}}(t)$. We consider two cases.

- **Case $x\notin\mathtt{fv}(t)$.** Then $\mathtt{mskel}^{\{x\}}(t)=t$ and $t\{x\leftarrow\overline{x^{\circ}}^{\Uparrow}\}=t$, so $\lambda^{\circ}x.t\{x\leftarrow\overline{x^{\circ}}^{\Uparrow}\}=\lambda^{\circ}x.t=\lambda^{\circ}x.\mathtt{mskel}^{\{x\}}(t)=\mathtt{mskel}(\lambda x.t)$. Thus, $\mathsf{init}(\lambda x.t)\to_{\mathtt{pr3}}\overline{\mathtt{mskel}(\lambda x.t)}^{\Uparrow}$. Moreover, $n=1=|\mathtt{mskel}(\lambda x.t)|_{\circ}$, since the only white construct of $\mathtt{mskel}(\lambda x.t)$ is the root abstraction.
- **Case $x\in\mathtt{fv}(t)$.** By Lemma 15, $\lambda^{\circ}x.t\{x\leftarrow\overline{x^{\circ}}^{\Uparrow}\}\to_{\mathtt{alg}}^{k}\lambda^{\circ}x.\overline{\mathtt{mskel}^{\{x\}}(t)}^{\Uparrow}$, where $k=|\mathtt{mskel}^{\{x\}}(t)|_{\circ}-|t|_{\circ}-1=|\mathtt{mskel}^{\{x\}}(t)|_{\circ}-1$. Therefore:

$$\begin{array}{rcccl}\mathsf{init}(\lambda x.t) & \to_{\mathtt{pr3}} & \overline{\lambda^{\circ}x.t\{x\leftarrow\overline{x^{\circ}}^{\Uparrow}\}}^{\Uparrow} & \to_{\mathtt{alg}}^{k} & \overline{\lambda^{\circ}x.\overline{\mathtt{mskel}^{\{x\}}(t)}^{\Uparrow}}^{\Uparrow}\\ & \to_{\mathtt{ab3}} & \overline{\lambda^{\circ}x.\mathtt{mskel}^{\{x\}}(t)}^{\Uparrow} & = & \overline{\mathtt{mskel}(\lambda x.t)}^{\Uparrow}.\end{array}$$

Now, $n=k+2=|\mathtt{mskel}^{\{x\}}(t)|_{\circ}-1+2=|\mathtt{mskel}^{\{x\}}(t)|_{\circ}+1=|\mathtt{mskel}(\lambda x.t)|_{\circ}$. ◀

5.1 Complexity Analysis

We proved that the number of rewriting steps of the algorithm for computing the marked skeleton $\mathtt{mskel}(v)$ is exactly its marked size $|\mathtt{mskel}(\lambda x.t)|_{\circ}$ (Theorem 16), which is the size of the skeleton (Prop. 12). The complexity of the algorithm is indeed linear but it requires some discussion, since $\to_{\mathtt{pr3}}$ steps are not constant time operations. Moreover, we need to take into account the cost of splitting.

Complexity Analysis 1: Propagation + Absorption. For computing $\texttt{mskel}(v)$ from v, we assume that abstractions have pointers to the occurrences of the variable, so that the substitution $m\{x \leftarrow \overline{x}^{\circ\Uparrow}\}$ in $\to_{\texttt{pr3}}$ does not require going through the whole of m. This is Shivers and Wand's key idea, and also how our implementation works (see Appendix A). A global argument gives the cost: the algorithm turns $|\texttt{mskel}(v)|_\circ$ unmarked constructs into $\circ$-constructs exactly once, and it generates at most $|\texttt{mskel}(v)|_\circ$ $\Uparrow$-constructs that are never duplicated and are absorbed exactly once, thus it globally requires $\mathcal{O}(|\texttt{mskel}(v)|_\circ)$ time.

▶ **Lemma 17** (Cost of propagation + absorption). *Computing* $\texttt{mskel}(v)$ *from* v *requires time* $\mathcal{O}(|\texttt{mskel}(v)|_\circ)$.

Complexity Analysis 2: Splitting. For the cost of splitting, Lemma 10 guarantees that the unmarked sub-terms of marked skeletons can be recognized in $\mathcal{O}(1)$ by only inspecting their topmost constructor, so that implementing the splitting function takes time proportional to the number of marked constructors only.

▶ **Lemma 18** (Cost of splitting). *Computing* $\texttt{split}(\texttt{mskel}(v))$ *from* $\texttt{mskel}(v)$ *requires time* $\mathcal{O}(|\texttt{mskel}(v)|_\circ)$.

The next theorem sums it all up, recasting the obtained costs in the unmarked setting via the soundness of the algorithm (Prop. 12).

▶ **Theorem 19** (The algorithm is linear in the skeleton). *Let* $v \in \Lambda$ *be a value. Then the skeletal decomposition* $(\texttt{skel}(v), \texttt{flesh}(v))$ *of* v *can be computed in* $\mathcal{O}(|\texttt{skel}(v)|)$.

6 Preliminaries about Abstract Machines

In this section, we introduce terminology and basic concepts about abstract machines, that shall be the topic of the following sections.

Abstract Machines Glossary. Abstract machines manipulate *pre-terms*, that is, terms without implicit α-renaming. In this paper, an *abstract machine* is a quadruple $\texttt{M} = (\texttt{States}, \rightsquigarrow, \cdot \triangleleft \cdot, \underline{\cdot})$ the components of which are as follows.

- *States.* A state $\texttt{Q} \in \texttt{States}$ is a quadruple $(\texttt{C}, t, \texttt{S}, \texttt{E})$ composed by the *active term* t, plus three data structure, namely the *chain* $\texttt{C}$, the *(applicative) stack* $\texttt{S}$, and the *(global) environment* $\texttt{E}$. Terms in states are actually pre-terms.
- *Transitions.* The pair $(\texttt{States}, \rightsquigarrow)$ is a transition system with transitions $\rightsquigarrow$ partitioned into *principal transitions*, whose union is noted $\rightsquigarrow_{\texttt{pr}}$ and that are meant to correspond to steps on the calculus, and *search transitions*, whose union is noted $\rightsquigarrow_{\texttt{sea}}$, that take care of searching for (principal) redexes.
- *Initialization.* The component $\triangleleft \subseteq \Lambda \times \texttt{States}$ is the *initialization relation* associating closed terms without ESs to initial states. It is a *relation* and not a function because $t \triangleleft \texttt{Q}$ maps a *closed* λ-term t (considered modulo α) to a state $\texttt{Q}$ having a *pre-term representant* of t (which is not modulo α) as active term. Intuitively, any two states $\texttt{Q}$ and $\texttt{Q}'$ such that $t \triangleleft \texttt{Q}$ and $t \triangleleft \texttt{Q}'$ are α-equivalent. A state $\texttt{Q}$ is *reachable* if it can be reached starting from an initial state, that is, if $\texttt{Q}' \rightsquigarrow^* \texttt{Q}$ where $t \triangleleft \texttt{Q}'$ for some t and $\texttt{Q}'$, shortened as $t \triangleleft \texttt{Q}' \rightsquigarrow^* \texttt{Q}$.
- *Read-back.* The read-back function $\underline{\cdot} : \texttt{States} \to \texttt{Terms}$ turns reachable states into terms (possibly with ESs, that is, $\texttt{Terms} = \Lambda_{\texttt{es}}$ or $\texttt{Terms} = \Lambda_{\texttt{sk}}$) and satisfies the *initialization constraint*: if $t \triangleleft \texttt{Q}$ then $\underline{\texttt{Q}} =_\alpha t$.

A state is *final* if no transitions apply. A *run* $r : \mathtt{Q} \rightsquigarrow^* \mathtt{Q}'$ is a possibly empty finite sequence of transitions, the length of which is noted $|r|$; note that the first and the last states of a run are not necessarily initial and final. If a and b are transitions labels (that is, $\rightsquigarrow_a \subseteq \rightsquigarrow$ and $\rightsquigarrow_b \subseteq \rightsquigarrow$) then $\rightsquigarrow_{a,b} := \rightsquigarrow_a \cup \rightsquigarrow_b$ and $|r|_a$ is the number of a transitions in r, $|r|_{a,b} := |r|_a + |r|_b$, and similarly for more than two labels.

For the machines at work in this paper, the pre-terms in initial states shall be *well-bound*, that is, they have pairwise distinct bound names; for instance $(\lambda x.x)\lambda y.y$ is well-bound while $(\lambda x.x)\lambda x.x$ is not. We shall also write t^α in a state $\mathtt{Q}$ for a *fresh well-bound renaming* of t, *i.e.* t^α is α-equivalent to t, well-bound, and its bound variables are fresh with respect to those in t and in the other components of $\mathtt{Q}$.

Mechanical Bismulations. Machines are usually showed to be correct with respect to a strategy via some form of bisimulation relating terms and machine states. The notion that we adopt is here dubbed *mechanical bisimulation*, and it should not be confused with the strong bisimulation property of structural equivalence (which actually plays a role in mechanical bisimulations). The definition, tuned towards complexity analyses, requires a perfect match between the steps of the evaluation sequence and the principal transitions of the machine run. *Terminology*: a structural strategy $(\to_{\mathtt{str}}, \equiv)$ is a strategy $\to_{\mathtt{str}}$ plus a relation $\equiv$ that is a strong bisimulation with respect to $\to_{\mathtt{str}}$ (see Prop. 4).

▶ **Definition 20** (Mechanical bisimulation). *A machine* $\mathtt{M} = (\mathtt{States}, \rightsquigarrow, \cdot \triangleleft \cdot, \underline{\cdot})$ *and a structural strategy* $(\to_{\mathtt{str}}, \equiv)$ *on* $\Lambda_{\mathtt{sk}}$*-terms are mechanical bisimilar when, given an initial state* $t \triangleleft \mathtt{Q}$*:*

1. Runs to evaluations*: for any run* $r : t \triangleleft \mathtt{Q} \rightsquigarrow^* \mathtt{Q}'$ *there exists an evaluation* $\mathtt{e} : t \to^*_{\mathtt{str}} \equiv \underline{\mathtt{Q}'}$*;*
2. Evaluations to runs*: for every evaluation* $\mathtt{e} : t \to^*_{\mathtt{str}} u$ *there exists a run* $r : t \triangleleft \mathtt{Q} \rightsquigarrow^* \mathtt{Q}'$ *such that* $\underline{\mathtt{Q}'} \equiv u$*;*
3. Principal matching*: for every principal transition* $\rightsquigarrow_{\mathtt{a}}$ *of label* a *of* $\mathtt{M}$*, in both previous points the number* $|r|_{\mathtt{a}}$ *of* $\mathtt{a}$*-transitions in* r *is exactly the number* $|\mathtt{e}|_a$ *of of* $\mathtt{a}$*-steps in the evaluation* $\mathtt{e}$*,* i.e. $|\mathtt{e}|_{\mathtt{a}} = |r|_{\mathtt{a}}$*.*

The proof that a machine and a strategy are in a mechanical bisimulation follows from some basic properties, grouped under the notion of distillery, following Accattoli et al. [4].

▶ **Definition 21** (Distillery). *A machine* $\mathtt{M} = (\mathtt{States}, \rightsquigarrow, \cdot \triangleleft \cdot, \underline{\cdot})$ *and a structural strategy* $(\to_{\mathtt{str}}, \equiv)$ *are a* distillery *if the following conditions hold:*

1. Principal projection*:* $\mathtt{Q} \rightsquigarrow_{\mathtt{a}} \mathtt{Q}'$ *implies* $\underline{\mathtt{Q}} \to_{\mathtt{a}} \equiv \underline{\mathtt{Q}'}$ *for every principal transition of label* $\mathtt{a}$*;*
2. Search transparency*:* $\mathtt{Q} \rightsquigarrow_{\mathtt{sea}} \mathtt{Q}'$ *implies* $\underline{\mathtt{Q}} = \underline{\mathtt{Q}'}$*;*
3. Search transitions terminate*:* $\rightsquigarrow_{\mathtt{sea}}$ *terminates;*
4. Determinism*:* $\to_{\mathtt{str}}$ *is deterministic;*
5. Halt*:* $\mathtt{M}$ *final states decode to* $\to_{\mathtt{str}}$*-normal terms.*

▶ **Theorem 22** (Sufficient condition for implementations). *Let a machine* $\mathtt{M}$ *and a structural strategy* $(\to_{\mathtt{str}}, \equiv)$ *be a distillery. Then, they are mechanical bisimilar.*

7 The MAD and the Skeletal MAD

In this section, we study an abstract machine for skeletal CbNeed, obtained as a minor modification of a known machine for CbNeed, that is recalled here. Since the two machines share most data structures and transitions, they are presented together in Fig. 6 (slightly abusing notations: the two machines define environments $\mathtt{E}$ differently, so the notation of the common transitions is overloaded).

$$
\begin{array}{rrcl}
\textsc{Stacks} & \mathtt{S} & ::= & \epsilon \mid t{:}\mathtt{S} \\
\textsc{MAD envs} & \mathtt{E} & ::= & \epsilon \mid [x{\leftarrow}t]{:}\mathtt{E} \\
\textsc{Skel. MAD envs} & \mathtt{E} & ::= & \epsilon \mid [x{\leftarrow}t]{:}\mathtt{E} \mid \langle x{\leftarrow}v\rangle{:}\mathtt{E}
\end{array}
\qquad
\begin{array}{rrcl}
\textsc{Chains} & \mathtt{C} & ::= & \epsilon \mid \mathtt{C}{:}(x,\mathtt{S},\mathtt{E}[x{\leftarrow}\cdot]) \\
\textsc{States} & \mathtt{Q} & := & (\mathtt{C},t,\mathtt{S},\mathtt{E}) \\
\textsc{Init.} & t & \triangleleft & (\epsilon, t^{\alpha}, \epsilon, \epsilon) \quad (\#)
\end{array}
$$

Chain	Code	Stack	Env		Chain	Code	Stack	Env	
Common transitions									
$\mathtt{C}$	tu	$\mathtt{S}$	$\mathtt{E}$	$\rightsquigarrow_{\mathtt{sea}_1}$	$\mathtt{C}$	t	$u{:}\mathtt{S}$	$\mathtt{E}$	
$\mathtt{C}$	$\lambda x.t$	$u{:}\mathtt{S}$	$\mathtt{E}$	$\rightsquigarrow_{\beta}$	$\mathtt{C}$	t	$\mathtt{S}$	$[x{\leftarrow}u]{:}\mathtt{E}$	
$\mathtt{C}$	x	$\mathtt{S}$	$\mathtt{E}{:}[x{\leftarrow}t]{:}\mathtt{E}'$	$\rightsquigarrow_{\mathtt{sea}_2}$	$\mathtt{C}{:}(x,\mathtt{S},\mathtt{E}[x{\leftarrow}\cdot])$	t	ϵ	$\mathtt{E}'$	$(*)$
$\mathtt{C}{:}(x,\mathtt{S},\mathtt{E}[x{\leftarrow}\cdot])$	v	ϵ	$\mathtt{E}'$	$\rightsquigarrow_{\mathtt{sea}_3}$	$\mathtt{C}$	x	$\mathtt{S}$	$\mathtt{E}{:}[x{\leftarrow}v]{:}\mathtt{E}'$	
MAD specific transition									
$\mathtt{C}$	x	$\mathtt{S}$	$\mathtt{E}{:}[x{\leftarrow}v]{:}\mathtt{E}'$	$\rightsquigarrow_{\mathtt{sub}}$	$\mathtt{C}$	v^{α}	$\mathtt{S}$	$\mathtt{E}{:}[x{\leftarrow}v]{:}\mathtt{E}'$	$(\#)$
Skeletal MAD specific transitions									
$\mathtt{C}$	x	$\mathtt{S}$	$\mathtt{E}{:}[x{\leftarrow}v]{:}\mathtt{E}'$	$\rightsquigarrow_{\mathtt{sk}}$	$\mathtt{C}$	x	$\mathtt{S}$	$\mathtt{E}{:}\langle x{\leftarrow}v'\rangle{:}\mathtt{E}_S{:}\mathtt{E}'$	$(\%)$
$\mathtt{C}$	x	$\mathtt{S}$	$\mathtt{E}{:}\langle x{\leftarrow}v\rangle{:}\mathtt{E}'$	$\rightsquigarrow_{\mathtt{ss}}$	$\mathtt{C}$	v^{α}	$\mathtt{S}$	$\mathtt{E}{:}\langle x{\leftarrow}v\rangle{:}\mathtt{E}'$	$(\#)$

$(\#)$ t^{α} is any well-bound code α-equivalent to t such that its bound names are fresh with respect to those in the rest of the state.
$(*)$ If t is not a value $(\%)$ Where $\mathtt{skdec}(v) = (v', S)$ is the skeletal decomposition of v (computed as $\mathtt{split}(\mathtt{mskel}(v))$), and $\mathtt{E}_S$ is S seen as an environment.

Read-back

$$
\begin{array}{rrcl}
\textsc{Empty} & \underline{\epsilon} & := & \langle\cdot\rangle \\
\textsc{Stacks} & \underline{t{:}\mathtt{S}} & := & \underline{\mathtt{S}}\langle\langle\cdot\rangle t\rangle \\
\textsc{Chains} & \underline{\mathtt{C}{:}(x,\mathtt{S},\mathtt{E}[x{\leftarrow}\cdot])} & := & \underline{\mathtt{E}}\langle\underline{\mathtt{C}}\langle\underline{\mathtt{S}}\langle x\rangle\rangle\rangle[x{\leftarrow}\langle\cdot\rangle]
\end{array}
\qquad
\begin{array}{rrcl}
\textsc{Envs} & \underline{[x{\leftarrow}t]{:}\mathtt{E}} & := & \underline{\mathtt{E}}\langle\langle\cdot\rangle[x{\leftarrow}t]\rangle \\
 & \underline{\langle x{\leftarrow}t\rangle{:}\mathtt{E}} & := & \underline{\mathtt{E}}\langle\langle\cdot\rangle\langle x{\leftarrow}t\rangle\rangle \\
\textsc{States} & \underline{(\mathtt{C},t,\mathtt{S},\mathtt{E})} & := & \underline{\mathtt{E}}\langle\underline{\mathtt{C}}\langle\underline{\mathtt{S}}\langle t\rangle\rangle\rangle
\end{array}
$$

Figure 6 The Milner Abstract machine by Need (MAD) and the Skeletal Milner Abstract machine by Need (Skeletal MAD) presented as different extension of a common set of transitions (but be careful: the common transitions use different notions of environment).

The MAD. The *Milner Abstract machine by neeD* (MAD) of Fig. 6 implements the CbNeed strategy $\rightarrow_{\mathtt{need}}$. It is a CbNeed variant of the *Milner Abstract Machine* (MAM), itself a simplification of the Krivine abstract machine (KAM). The MAM is a CbN abstract machine using a single global environment (sometimes called *heap*), a stack for arguments, and no closures – it is omitted here (the KAM instead uses closures and many local environments). Both the MAM and the MAD are introduced by Accattoli et al. in [4].

The MAD modifies the MAM by adding a mechanism realizing the memoization characteristic of CbNeed. Namely, when execution encounters a variable occurrence x associated to an environment entry $[x{\leftarrow}u]$ such that u is not a value, then the MAM duplicates u while the MAD does not. Instead, part of the current state is saved on the new *chain* data structure $\mathtt{C}$, and the MAD starts evaluating u inside the environment entry itself; this is transition $\rightsquigarrow_{\mathtt{sea}_2}$. When such an evaluations ends with a value v, the machines resumes the state saved on the chain, and the environment is updated replacing u with v – this is transition $\rightsquigarrow_{\mathtt{sea}_3}$ – so that future encounters of x will not need to re-evaluate u. Both the MAM and the MAD rely on α-renaming as a black-box operation $\cdot^{\alpha}$, at work in transition $\rightsquigarrow_{\mathtt{sub}}$.

The presentation in Fig. 6 differs from the MAD in [4] in a very minor point. In [4], transitions $\rightsquigarrow_{\mathtt{sea}_3}$ and $\rightsquigarrow_{\mathtt{sub}}$ are concatened into a single transition. Splitting them helps in stating the invariants of the machines.

The Skeletal MAD. The Skeletal MAD, also in Fig. 6, modifies the MAD exactly as the Skeletal CbNeed LSC modifies the CbNeed LSC, that is, by adding skeletal entries $\langle x{\leftarrow}v\rangle$ to the global environment and splitting the substitution transtition $\rightsquigarrow_{\mathtt{sub}}$ into two: the skeletonizing transition $\rightsquigarrow_{\mathtt{sk}}$ that separates the skeleton v' from the flesh of the value v, and the skeletal substitution transition $\rightsquigarrow_{\mathtt{ss}}$ that substitutes the (α-renamed) skeleton v'^{α}.

Notation. For specifying transition $\rightsquigarrow_{\mathtt{sk}}$, we use the following notation for the flesh: a substitution context $S = \langle\cdot\rangle[\![x_1{\leftarrow}t_1]\!]\dots[\![x_k{\leftarrow}t_k]\!]$, where each $[\![x_i{\leftarrow}t_i]\!]$ can be $[x_i{\leftarrow}t_i]$ or $\langle x_i{\leftarrow}t_i\rangle$, induces a global environment $\mathtt{E}_S := [\![x_1{\leftarrow}t_1]\!]{:}\dots{:}[\![x_k{\leftarrow}t_k]\!]{:}\epsilon$.

The union of all the transition rules of the Skeletal MAD (namely, $\mathtt{sea}_1$, $\mathtt{sea}_2$, $\mathtt{sea}_3$, β, $\mathtt{sk}$, and $\mathtt{ss}$) is noted $\rightsquigarrow_{\mathtt{SMAD}}$.

The read-back of (Skeletal) MAD states to terms (possibly with ESs) is defined in Fig. 6. The definition uses auxiliary notions of read-back for chains, stacks, and environments that turn them into CbNeed evaluation contexts. For stacks and environments, their read-back is clearly an evaluation context. For chains, it shall be proved as an invariant of the machine.

Invariants. To prove properties of the Skeletal MAD we need to isolate some of its invariants in Lemma 24 below. The closure one ensures that the closure of the initial term extends, in an appropriate sense, to all reachable states. The well-bound invariant, similarly, ensures that binders in reachable states are pairwise distinct, as in the initial term. To compactly formulate the closure invariant we need the notion of terms of a state.

▶ **Definition 23** (Terms of a state). *Let* $\mathtt{Q} = (\mathtt{C}, u, \mathtt{S}, \mathtt{E})$ *be a Skeletal MAD state where* $\mathtt{C}$ *is a possibly empty chain* $\epsilon{:}(x_k, \mathtt{S}_k, \mathtt{E}_k{:}[x_k{\leftarrow}\cdot]){:}\dots{:}(x_1, \mathtt{S}_1, \mathtt{E}_1{:}[x_1{\leftarrow}\cdot])$ *for some* $k \geq 0$*. The* terms of $\mathtt{Q}$ *are* u*, every term in* $\mathtt{S}$ *and* $\mathtt{S}_i$*, and every term in an entry of* $\mathtt{E}$ *or* $\mathtt{E}_i$*, for* $1 \leq i \leq k$*.*

▶ **Lemma 24** (Qualitative invariants). *Let* $t \triangleleft \mathtt{Q} \rightsquigarrow^* \mathtt{Q}' = (\mathtt{C}, u, \mathtt{S}, \mathtt{E})$ *be a Skeletal MAD run.*

1. Closure*: for every term* s *of* $\mathtt{Q}'$ *and for any variable* $x \in \mathtt{fv}(s)$ *there is an environment entry* $[x{\leftarrow}t]$*,* $\langle x{\leftarrow}v\rangle$*, or* $[x{\leftarrow}\cdot]$ *on the right of* s *in* $\mathtt{Q}'$*.*
2. Well-bound*: if* $\lambda x.s$ *occurs in* $\mathtt{Q}'$ *and* x *has any other occurrence in* $\mathtt{Q}'$ *then it is as a free variable of* s*, and for any substitution* $[y{\leftarrow}t]$*,* $\langle y{\leftarrow}v\rangle$*, or* $[y{\leftarrow}\cdot]$ *in* $\mathtt{Q}'$ *the name* y *can occur (in any form) only on the left of that entry in* $\mathtt{Q}'$*.*
3. Contextual chain read-back*:* $\underline{\mathtt{C}}$*,* $\underline{\mathtt{C}\langle\mathtt{S}\rangle}$*,* $\underline{\mathtt{E}\langle\mathtt{C}\rangle}$ *and* $\underline{\mathtt{E}\langle\mathtt{C}\langle\mathtt{S}\rangle\rangle}$ *are CbNeed evaluation contexts.*

Mechanical Bisimulation. The invariants allow us to prove that the Skeletal MAD and the strategy $\rightarrow_{\mathtt{sk{\cdot}need}}$ modulo $\equiv$ form a distillery, from which the mechanical bisimulation follows (by Theorem 22). The closure invariant is used for Point 4, the other two for Point 1.

▶ **Theorem 25** (Skeletal distillery). *The Skeletal MAD and the structural strategy* $(\rightarrow_{\mathtt{sk{\cdot}need}}, \equiv)$ *form a distillery. Namely, let* $\mathtt{Q}$ *be a reachable Skeletal MAD state.*

1. Principal projection*:*
   a. *if* $\mathtt{Q} \rightsquigarrow_{\beta} \mathtt{Q}'$ *then* $\underline{\mathtt{Q}} \rightarrow_{\mathtt{sk{\cdot}dB}} \equiv \underline{\mathtt{Q}'}$*.*
   b. *if* $\mathtt{Q} \rightsquigarrow_{\mathtt{sk}} \mathtt{Q}'$ *then* $\underline{\mathtt{Q}} \rightarrow_{\mathtt{sk}} \underline{\mathtt{Q}'}$*.*
   c. *if* $\mathtt{Q} \rightsquigarrow_{\mathtt{ss}} \mathtt{Q}'$ *then* $\underline{\mathtt{Q}} \rightarrow_{\mathtt{ss}} \underline{\mathtt{Q}'}$*.*
2. Search transparency*: if* $\mathtt{Q} \rightsquigarrow_{\mathtt{sea}_1,\mathtt{sea}_2,\mathtt{sea}_3} \mathtt{Q}'$ *then* $\underline{\mathtt{Q}} = \underline{\mathtt{Q}'}$*.*
3. Search terminates*:* $\rightsquigarrow_{\mathtt{sea}_1,\mathtt{sea}_2,\mathtt{sea}_3}$ *is terminating.*
4. Halt*: if* $\mathtt{Q}$ *is final then it has shape* $(\epsilon, v, \epsilon, \mathtt{E})$ *and* $\underline{\mathtt{Q}}$ *is* $\rightarrow_{\mathtt{sk{\cdot}need}}$*-normal.*

▶ **Corollary 26** (Skeletal mechanical bisimulation). *The Skeletal MAD and the structural strategy* $(\rightarrow_{\mathtt{sk{\cdot}need}}, \equiv)$ *are in a mechanical bisimulation.*

8 Complexity Analysis

In this section, we show that the Skeletal MAD can be concretely implemented within the same complexity of the MAD, that is, with *bi-linear* overhead.

Sub-Term Property. As it is standard, the complexity analysis crucially relies on the sub-term property, that bounds the size of manipulated (and thus duplicated) terms using the size of the initial term. The proof requires a similar property about skeletons.

▶ **Lemma 27** (Skeleton and flesh size).
1. Auxiliary parametrized statement*: let* $t \in \Lambda$, $\mathtt{V}$ *be a set of variables, and* $\mathtt{skdec}^{\mathtt{V}}(t) := (u, S)$*. Then* $|u| \leq |t|$ *and* $|s| \leq |t|$ *for any ES* $[x \leftarrow s]$ *in* S*;*
2. Main*: let* $v \in \Lambda$*. Then* $|\mathtt{skel}(v)| \leq |v|$ *and* $|s| \leq |v|$ *for any ES* $[x \leftarrow s]$ *in* $\mathtt{flesh}(v)$*.*

▶ **Lemma 28** (Quantitative sub-term property). *Let* $t \triangleleft \mathtt{Q} \rightsquigarrow^{*}_{\mathtt{SMAD}} \mathtt{Q}'$ *be a Skeletal MAD run. Then* $|u| \leq |t|$ *for any term* u *of* $\mathtt{Q}'$*.*

Next, some basic observations about the transitions, together with the sub-term property, allow us to bound their number using the two key parameters (that is, number of β-steps/transitions and size of the initial term).

▶ **Proposition 29** (Number of transitions). *Let* $r : t \triangleleft \mathtt{Q} \rightsquigarrow^{*}_{\mathtt{SMAD}} \mathtt{Q}'$ *be a Skeletal MAD run.*
1. $|r|_{\mathtt{sk},\mathtt{ss},\mathtt{sea}_2,\mathtt{sea}_3} \in \mathcal{O}(|r|_\beta)$*, and*
2. $|r|_{\mathtt{sea}_1} \in \mathcal{O}(|t| \cdot (|r|_\beta + 1))$*.*

Proof.
1. **a.** $|r|_{\mathtt{sea}_2} \leq |r|_\beta$, because every $\mathtt{sea}_2$ transition consumes one a non-value entry from the environment, which are created only by β transitions.
   b. $|r|_{\mathtt{sea}_3} \leq |r|_\beta$, because every $\mathtt{sea}_3$ transition consumes one entry from the chain, which are created only by $\mathtt{sea}_2$ transitions, which in turn are bound by β transitions.
   c. $|r|_{\mathtt{ss}} \in \mathcal{O}(|r|_\beta)$, because after a $\mathtt{ss}$ transition there can be either a $\mathtt{sea}_3$ transition, a β transition, or no transition (if the $\mathtt{ss}$ transition is the last one of the run r). Therefore, $|r|_{\mathtt{ss}} \leq |r|_{\mathtt{sea}_3} + |r|_\beta + 1 \leq 2|r|_\beta + 1$.
   d. $|r|_{\mathtt{sk}} \in \mathcal{O}(|r|_\beta)$, because every $\mathtt{sk}$ transition is followed by a $\mathtt{ss}$ transition.
2. Note that $\mathtt{sea}_1$ transitions decrease the size of the code, which is increased only by $\mathtt{ss}$ and $\mathtt{sea}_2$ transition. By the sub-term property (Lemma 28), the code increase is bounded by the size $|t|$ of the initial term. By Point 1, $|r|_{\mathtt{ss},\mathtt{sea}_2} = \mathcal{O}(|r|_\beta)$. Then $|r|_{\mathtt{sea}_1} \in \mathcal{O}(|t| \cdot (|r|_{\mathtt{ss},\mathtt{sea}_2} + 1)) = \mathcal{O}(|t| \cdot (|r|_\beta + 1))$. ◀

Lastly, we need some assumptions on how the Skeletal MAD can be concretely implemented. Our OCaml implementation (see Appendix A) represents variables as memory locations and variable occurrences as pointers to those locations, obtaining random access to environment entries in $\mathcal{O}(1)$. As already mentioned in Sect. 5, all constructors have pointers to their parents in the syntactic tree, and in particular variables have pointers to their occurrences, as to implement efficiently the computation of the skeleton. Moreover, environments are implemented as doubly linked lists, to split/concatenate them in $\mathcal{O}(1)$ in transitions $\mathtt{sea}_2$ and $\mathtt{sea}_3$. The cost of transitions $\mathtt{sk}$ and $\mathtt{ss}$ is bound by the size of the value to skeletonize/copy, which is bound by the sub-term property.

▶ **Lemma 30** (Cost of single transitions). *Let* $t \triangleleft \mathtt{Q} \rightsquigarrow^{*}_{\mathtt{SMAD}} \mathtt{Q}'$ *be a run. Implementing any transitions from* $\mathtt{Q}'$ *costs* $\mathcal{O}(1)$ *but for* $\mathtt{sk}$ *and* $\mathtt{ss}$ *transitions, which cost* $\mathcal{O}(|t|)$ *each.*

Putting all together, we obtain a bilinear bound.

▶ **Theorem 31** (The Skeletal MAD is bi-linear). *Let* $r : t \triangleleft \mathtt{Q} \rightsquigarrow^{*}_{\mathtt{SMAD}} \mathtt{Q}'$ *be a Skeletal MAD run. Then* r *can be implemented on random access machiness in* $\mathcal{O}(|t| \cdot (|r|_\beta + 1))$*.*

9 Conclusions

We show that skeletal call-by-need provides, in some cases, exponentially shorter evaluation sequences involving exponentially smaller terms than call-by-need. We also show that the required skeleton reconstruction can be implemented in linear time and space, by giving a new simpler and graph-free presentation of ideas by Shivers and Wand. Lastly, we smoothly plug this result in the existing distillation technique for abstract machines, obtaining an implementation of skeletal call-by-need with bi-linear overhead. In particular, the bi-linear overhead allows us to establish that the shrinking of evaluation sequences is a real time speed-up: the smart specification of skeletal call-by-need does not hide an expensive overhead.

References

1 Beniamino Accattoli. An abstract factorization theorem for explicit substitutions. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, volume 15 of *LIPIcs*, pages 6–21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPICS.RTA.2012.6.

2 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.

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

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

5 Beniamino Accattoli and Bruno Barras. Environments and the complexity of abstract machines. In Wim Vanhoof and Brigitte Pientka, editors, *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017*, pages 4–16. ACM, 2017. doi:10.1145/3131851.3131855.

6 Beniamino Accattoli, Eduardo Bonelli, Delia Kesner, and Carlos Lombardi. A nonstandard standardization theorem. In Suresh Jagannathan and Peter Sewell, editors, *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. ACM, 2014. doi:10.1145/2535838.2535886.

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

8 Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. Strong call-by-value is reasonable, implosively. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–14. IEEE, 2021. doi:10.1109/LICS52264.2021.9470630.

9 Beniamino Accattoli and Ugo Dal Lago. On the invariance of the unitary cost model for head reduction. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, volume 15 of *LIPIcs*, pages 22–37. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPICS.RTA.2012.22.

10 Beniamino Accattoli and Ugo Dal Lago. (leftmost-outermost) beta reduction is invariant, indeed. *Log. Methods Comput. Sci.*, 12(1), 2016. doi:10.2168/LMCS-12(1:4)2016.

11 Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. The (in)efficiency of interaction. *Proc. ACM Program. Lang.*, 5(POPL):1–33, 2021. doi:10.1145/3434332.

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

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

14 Beniamino Accattoli and Delia Kesner. The structural λ-calculus. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, 24th International Workshop , CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings*, volume 6247 of *Lecture Notes in Computer Science*, pages 381–395. Springer, 2010. doi:10.1007/978-3-642-15205-4_30.

15 Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. Reasonable space for the λ-calculus, logarithmically. *Log. Methods Comput. Sci.*, 20(4), 2024. doi:10.46298/LMCS-20(4:15)2024.

16 Beniamino Accattoli and Adrienne Lancelot. Mirroring call-by-need, or values acting silly. In Jakob Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, July 10-13, 2024, Tallinn, Estonia*, volume 299 of *LIPIcs*, pages 23:1–23:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.FSCD.2024.23.

17 Beniamino Accattoli and Maico Leberle. Useful open call-by-need. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany*, volume 216 of *LIPIcs*, pages 4:1–4:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CSL.2022.4.

18 Beniamino Accattoli, Francesco Magliocca, Loïc Peyrot, and Claudio Sacerdoti Coen. The cost of skeletal call-by-need, smoothly, 2025. arXiv:2505.09242.

19 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.

20 Beniamino Accattoli and Claudio Sacerdoti Coen. IMELL cut elimination with linear overhead. In Jakob Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, July 10-13, 2024, Tallinn, Estonia*, volume 299 of *LIPIcs*, pages 24:1–24:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.FSCD.2024.24.

21 Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 233–246. ACM Press, 1995. doi:10.1145/199448.199507.

22 Thibaut Balabonski. A unified approach to fully lazy sharing. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 469–480. ACM, 2012. doi:10.1145/2103656.2103713.

23 Thibaut Balabonski. Weak optimality, and the meaning of sharing. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 263–274. ACM, 2013. doi:10.1145/2500365.2500606.

24 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.

25 Thibaut Balabonski, Antoine Lanco, and Guillaume Melquiond. A strong call-by-need calculus. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPIcs*, pages 9:1–9:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.FSCD.2021.9.

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

27 Andrea Condoluci, Beniamino Accattoli, and Claudio Sacerdoti Coen. Sharing equality is linear. In Ekaterina Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, pages 9:1–9:14. ACM, 2019. doi:10.1145/3354166.3354174.

28 Yannick Forster, Fabian Kunze, and Marc Roth. The weak call-by-value λ-calculus is reasonable for both time and space. *Proc. ACM Program. Lang.*, 4(POPL):27:1–27:23, 2020. doi:10.1145/3371095.

29 Tom Gundersen, Willem Heijltjes, and Michel Parigot. Atomic lambda calculus: A typed lambda-calculus with explicit sharing. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 311–320. IEEE Computer Society, 2013. doi:10.1109/LICS.2013.37.

30 John Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Programming Research Group, Oxford University, July 1983.

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

32 Delia Kesner, Loïc Peyrot, and Daniel Ventura. Node replication: Theory and practice. *Log. Methods Comput. Sci.*, 20(1), 2024. doi:10.46298/LMCS-20(1:5)2024.

33 Delia Kesner, Alejandro Ríos, and André s Viso. Call-by-need, neededness and all that. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Thessaloniki, Greece, April 14-20, 2018 , Proceedings*, volume 10803 of *Lecture Notes in Computer Science*, pages 241–257. Springer, 2018. doi:10.1007/978-3-319-89366-2_13.

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

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

36 Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

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

38 Peter Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3):231–264, 1997. doi:10.1017/S0956796897002712.

39 Olin Shivers and Mitchell Wand. Bottom-up beta-reduction: Uplinks and lambda-dags. *Fundam. Informaticae*, 103(1-4):247–287, 2010. doi:10.3233/FI-2010-328.

40 D. A. Turner. A new implementation technique for applicative languages. *Softw. Pract. Exp.*, 9(1):31–49, 1979. doi:10.1002/SPE.4380090105.

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

A OCaml Implementation

We provide at `https://github.com/Franciman/fullylazymad` a reference implementation in OCaml of the Skeletal MAD, meant to guarantee the soundness of the assumptions that underly the complexity analyses. The implementation takes in input a user provided term and it shows the full run printing all transitions and all intermediate machine states. We highlight here a few details on the implementation.

Data structures. Pre-terms are represented by the following algebraic data type (ADT):

```
type term =
  | Var of { v: var_ref; mutable taken: bool; mutable parent: term option }
  | Abs of { v: var_ref; mutable body: term; mutable occurrences: term list; mutable taken: bool; mutable
      parent: term option }
  | App of { mutable head: term; mutable arg: term; mutable taken: bool; mutable parent: term option }
and var_ref =
 { mutable prev : var_ref option;
   orig_name : string; (* to help generating new names *)
   name : string;
   mutable sub : sub;
   mutable next : var_ref option }
and sub =
  NoSub | Sub of term | SubSkel of term | Hole | Copy of (term list ref * var_ref)
```

whose constructors `App` for applications, `Abs` for abstractions and `Var` for variable occurrence hold mutable pointers to their subterms, a mutable optional pointer to the parent and a mutable boolean `taken` for marking terms during skeleton extraction.

Variable occurrences point to a shared, unique variable declaration/substitution in memory of type `var_ref`, which holds the variable name, a substitution description field of type `sub` and two optional pointers (`prev` and `next`) to insert the variable in an environment, that is a double-linked lists of variable declarations. A substitution description is an ADT whose constructors are `Sub` t for the explicit substitution $[x{\leftarrow}t]$, `SubSkel` t for the skeletal substitutions $\langle x{\leftarrow}v\rangle$, `NoSub` for bound variables, `Hole` to represent $[x{\leftarrow}\cdot]$ in chain items and `Copy (parents,`x`)` to point to another variable declaration x together with the list of its parents. The latter form is used temporarily to preserve sharing of variable declarations during the implementation of $\cdot^{\alpha}$: the first time a variable is to be copied its substitution is make to point to the new copy and each time an occurrence of the variable is to be copied, the same variable declaration is reused and a new parent is added for it.

Abstractions also point to the list of occurrences of the bound variable, to implement rule $\mapsto_{\mathsf{pr3}}$ in $\mathcal{O}(1)$. This is a minor memory-saving divergence from the paper where it was suggested to have backpointers to the occurrences stored in the bound variable declaration node, of type `var_ref`: once an abstraction is consumed, the backpointers become useless and thus avoiding them in the implementation saves some space.

Machine states are implemented straightforwardly using lists for stacks, chains, and the set of roots (which is then actually represented as a list, not as a set), and doubly linked lists of variable declarations for environments:

```
type env = var_ref option
  (* Some head of the doubly-linked list, or None for empty list *)
type stack = term list
type chain = (var_ref * stack * env) list
type state = chain * term * stack * env
```

Skeleton Reconstruction. The rewriting system for marking terms to extract the skeleton and flesh is implemented sequentially as a recursive function. Each activation record for a call to `mark_skeleton` in the OCaml stack corresponds in the paper to an occurrence of $\dot{\cdot}^{\Uparrow}$ in the rewritten term.

```
let rec mark_skeleton =
 function
  | None -> ()
  | Some (Var v) -> (v.taken <- true; mark_skeleton v.parent)
  | Some (Abs a) ->
      if not a.taken
      then (a.taken <- true;
            List.iter (fun v -> mark_skeleton (Some v)) a.occurrences;
            mark_skeleton a.parent)
  | Some (App a) ->
      if not a.taken
      then (a.taken <- true; mark_skeleton a.parent)
```

Parallel implementations of `mark_skeleton` are possible, but because of the necessity of syncronization over application nodes and because the size of skeletonized terms is supposed to be small, it is unclear if actual parallelization would provide any speed-up.

Remaining Code. The code to extract the skeleton and flesh from marked terms, and the code for machine transitions follow straightforwardly the rules in the paper, up to noisy code to keep the doubly linked structure of terms and environments. Here is an example of the code for one transition in the Skeletal MAD:

```
let step : state -> string * state =
 function
  ...
  | chain, Abs { v; body; _ }, arg :: args, env ->
      set_parent body None;  (* unliks body from its parent (double link severed)*)
      v.sub <- Sub arg;      (* turns v into an explicit substitution *)
      push v env;            (* pushes v on top of the env *)
      "β",(chain, body, args, Some v)
  ...
```

The code that implements $\cdot^{\alpha}$ also follows the standard algorithm to copy a graph in linear time preserving the sharing, described by Accattoli and Barras in [5]. It uses the `Copy` constructor to temporarily associate each variable to its copy.

B Example of Skeletal MAD Run

We show here an example of invocation of our implementation. Suppose that the file `family_3.lambda` contains the following input that encodes the pure term that yields the third term of the family we studied, where backslashes are ASCII art equivalents of λ.

```
(\i. (\g. (\z. (z i) (z i)) (g (g (g i)))) (\x.\y. (x i) (x i) y)) (\w. w)
```

The command

```
dune exec bin/main.exe fully-lazy-functional-linked-env family_3.lambda
```

yields the following output[1]:

[1] The option `fully-lazy-functional-linked-env` selects the Skeletal MAD implementation described in the paper, that stays as close as possible to the abstract machine description. The repository also contains code for alternative implementations of the Skeletal MAD, comprising versions that keep the machine state garbage free.

$$
\begin{array}{ll}
 & |\underline{(\lambda i.(\lambda g.(\lambda z.zi(zi))(g(g(gi))))(\lambda x.\lambda y.xi(xi)y))(\lambda w.w)}||| \\
\to_{sea_1} & |\underline{\lambda i.(\lambda g.(\lambda z.zi(zi))(g(g(gi))))(\lambda x.\lambda y.xi(xi)y)}|(\lambda w.w)|| \\
\to_{\beta} & |\underline{(\lambda g.(\lambda z.zi(zi))(g(g(gi))))(\lambda x.\lambda y.xi(xi)y)}||[i{\leftarrow}\lambda w.w]| \\
\to_{sea_1} & |\underline{\lambda g.(\lambda z.zi(zi))(g(g(gi)))}|(\lambda x.\lambda y.xi(xi)y)|[i{\leftarrow}\lambda w.w]| \\
\to_{\beta} & |\underline{(\lambda z.zi(zi))(g(g(gi)))}||[g{\leftarrow}\lambda x.\lambda y.xi(xi)y] : [i{\leftarrow}\lambda w.w]| \\
\to_{sea_1} & |\underline{\lambda z.zi(zi)}|g(g(gi))|[g{\leftarrow}\lambda x.\lambda y.xi(xi)y] : [i{\leftarrow}\lambda w.w]| \\
\to_{\beta} & |\underline{zi(zi)}||[z{\leftarrow}g(g(gi))] : [g{\leftarrow}\lambda x.\lambda y.xi(xi)y] : [i{\leftarrow}\lambda w.w]| \\
\to_{sea_1} & |\underline{zi}|zi|[z{\leftarrow}g(g(gi))] : [g{\leftarrow}\lambda x.\lambda y.xi(xi)y] : [i{\leftarrow}\lambda w.w]| \\
\to_{sea_1} & |\underline{z}|i : zi|[z{\leftarrow}g(g(gi))] : [g{\leftarrow}\lambda x.\lambda y.xi(xi)y] : [i{\leftarrow}\lambda w.w]| \\
\to_{sea_2} & (z, i : zi, [z{\leftarrow}.])|\underline{g(g(gi))}||[g{\leftarrow}\lambda x.\lambda y.xi(xi)y] : [i{\leftarrow}\lambda w.w]| \\
\to_{sea_1} & (z, i : zi, [z{\leftarrow}.])|\underline{g}|g(gi)|[g{\leftarrow}\lambda x.\lambda y.xi(xi)y] : [i{\leftarrow}\lambda w.w]| \\
\to_{sk} & (z, i : zi, [z{\leftarrow}.])|\underline{g}|g(gi)|\langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{ss} & (z, i : zi, [z{\leftarrow}.])|\underline{\lambda x_1.\lambda y_2.x_1i(x_1i)y_2}|g(gi)|\langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{\beta} & (z, i : zi, [z{\leftarrow}.])|\underline{\lambda y_2.x_1i(x_1i)y_2}||[x_1{\leftarrow}g(gi)] : \langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{sea_3} & |\underline{z}|i : zi|[z{\leftarrow}\lambda y_2.x_1i(x_1i)y_2] : [x_1{\leftarrow}g(gi)] : \langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{sk} & |\underline{z}|i : zi|\langle z{\leftarrow}\lambda y_2.p_3y_2\rangle : [p_3{\leftarrow}x_1i(x_1i)] : [x_1{\leftarrow}g(gi)] : \langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{ss} & |\underline{\lambda y_4.p_3y_4}|i : zi|\langle z{\leftarrow}\lambda y_2.p_3y_2\rangle : [p_3{\leftarrow}x_1i(x_1i)] : [x_1{\leftarrow}g(gi)] : \langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{\beta} & |\underline{p_3y_4}|zi|[y_4{\leftarrow}i] : \langle z{\leftarrow}\lambda y_2.p_3y_2\rangle : [p_3{\leftarrow}x_1i(x_1i)] : [x_1{\leftarrow}g(gi)] : \langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{sea_1} & |\underline{p_3}|y_4 : zi|[y_4{\leftarrow}i] : \langle z{\leftarrow}\lambda y_2.p_3y_2\rangle : [p_3{\leftarrow}x_1i(x_1i)] : [x_1{\leftarrow}g(gi)] : \langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{sea_2} & (p_3, y_4 : zi, [y_4{\leftarrow}i] : \langle z{\leftarrow}\lambda y_2.p_3y_2\rangle : [p_3{\leftarrow}.])|\underline{x_1i(x_1i)}||[x_1{\leftarrow}g(gi)] : \langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{sea_1} & (p_3, y_4 : zi, [y_4{\leftarrow}i] : \langle z{\leftarrow}\lambda y_2.p_3y_2\rangle : [p_3{\leftarrow}.])|\underline{x_1i}|x_1i|[x_1{\leftarrow}g(gi)] : \langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{sea_1} & (p_3, y_4 : zi, [y_4{\leftarrow}i] : \langle z{\leftarrow}\lambda y_2.p_3y_2\rangle : [p_3{\leftarrow}.])|\underline{x_1}|i : x_1i|[x_1{\leftarrow}g(gi)] : \langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{sea_2} & (p_3, y_4 : zi, [y_4{\leftarrow}i] : \langle z{\leftarrow}\lambda y_2.p_3y_2\rangle : [p_3{\leftarrow}.]) : (x_1, i : x_1i, [x_1{\leftarrow}.])|\underline{g(gi)}||\langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{sea_1} & (p_3, y_4 : zi, [y_4{\leftarrow}i] : \langle z{\leftarrow}\lambda y_2.p_3y_2\rangle : [p_3{\leftarrow}.]) : (x_1, i : x_1i, [x_1{\leftarrow}.])|\underline{g}|gi|\langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{ss} & (p_3, y_4 : zi, [y_4{\leftarrow}i] : \langle z{\leftarrow}\lambda y_2.p_3y_2\rangle : [p_3{\leftarrow}.]) : (x_1, i : x_1i, [x_1{\leftarrow}.])|\underline{\lambda x_5.\lambda y_6.x_5i(x_5i)y_6}|gi|\langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\to_{\beta} & (p_3, y_4 : zi, [y_4{\leftarrow}i] : \langle z{\leftarrow}\lambda y_2.p_3y_2\rangle : [p_3{\leftarrow}.]) : (x_1, i : x_1i, [x_1{\leftarrow}.])|\underline{\lambda y_6.x_5i(x_5i)y_6}||[x_5{\leftarrow}gi] : \langle g{\leftarrow}\lambda x.\lambda y.xi(xi)y\rangle : [i{\leftarrow}\lambda w.w]| \\
\ldots &
\end{array}
$$

Number of betas: 24
Interpreter fully-lazy-functional-linked-env, final result: $\lambda w.w$

where in each machine state the chain is red, the code is underlined, the stack has no special marking and the environemnt is green.