Introduction to Abstract Machines and Their Complexity Analyses

MPRI course
Logique Linéaire et Paradigmes Logiques du Calcul,
Year 2023-24, part 4, Part 3, Lecture 3

Beniamino Accattoli

March 8, 2025

The purpose of this lecture is twofold: introducing abstract machines as well as their complexity analyses, in order to provide a complete proof of the reasonability of weak head reduction. The previous lecture explained why all proofs of reasonability for the abstract time cost model (given by a generic strategy $\to_{\mathtt{str}}$) are obtained via a formalism with sharing $(S, \to_{\mathtt{sh}X})$. Such proofs amount to proving that the two arrows with '?' in the following diagram can be realized within a polynomial overhead:

$$\begin{array}{ccc} (\Lambda, \to_{\mathtt{str}}) & \xleftarrow{\quad linear \quad} & \text{TMs} \\ ? \big\downarrow & & \big\uparrow quadratic \\ (S, \to_{\mathtt{sh}X}) & \xrightarrow[?]{\qquad\qquad} & \text{RAMs} \end{array}$$

The previous lecture sketched a proof for weak head reduction, using the linear substitution calculus (of which we only defined a specific strategy, micro weak head reduction) as a formalism for sharing. In this lecture, we provide a detailed proof of the overhead for the two arrows when $(S, \to_{\mathtt{sh}X})$ is an abstract machine, and in particular we shall introduce and analyze the *Milner abstract machine* (shortened to MAM). Additionally, we define the MAM via a principled introduction to abstract machines.

The lecture ends with the sketch of an OCaml implementation of the Milner abstract machine respecting the developed complexity analysis. In one of the next lectures, we shall relate the MAM with micro weak head reduction.

1 Weak Call-by-Name

We briefly recall the weak call-by-name (CbN) strategy, also known as weak head strategy or reduction (we use *reduction* and *strategy* as synonymous). It is defined as follows:

$$\frac{}{(\lambda x.t)u \to_{wh} t\{x \leftarrow u\}}\ (\beta) \qquad \frac{t \to_{wh} u}{ts \to_{wh} us}\ (@l) \tag{1}$$

This is the simplest possible evaluation strategy. Of course, it is deterministic. Let us mention two other ways of defining it, as they are going to be useful in the sequel:

1. *Synthetic rule*: the given inductive definition can be unfolded into a single synthetic rule
$$(\lambda x.t)us_1 \dots s_k \to_{wh} t\{x \leftarrow u\}s_1 \dots s_k \qquad k \geq 0$$
2. *Evaluation contexts*: define applicative (evaluation) contexts as
$$\text{APPLICATIVE CONTEXTS} \qquad A \;::=\; \langle\cdot\rangle \mid As$$
and define $\to_{wh}$ as
$$A\langle(\lambda x.t)u\rangle \to_{wh} A\langle t\{x \leftarrow u\}\rangle$$
where $A\langle t\rangle$ is the plugging t in the context A, amounting to replacing the hole $\langle\cdot\rangle$ with t, formally defined as $\langle\cdot\rangle\langle t\rangle := t$ and $(As)\langle t\rangle := A\langle t\rangle s$.

Weak Head Normal Forms. Normal forms with respect to $\to_{wh}$ have two possible shapes, they are either abstractions or terms of the form $xt_1 \dots t_k$ with $k \geq 0$, where t_i is whatever term, that is, it need not be normal. Sometimes, terms are required to be closed and in that cases weak head normal forms are simply abstractions.

2 Introducing Abstract Machines

Here we give a gentle introduction to abstract machines, starting from basic, first principles. To stress the generality and the modularity of the approach, we often abstract away from the weak CbN strategy and rather consider a generic strategy $\to_{\mathtt{str}}$.

Tasks of Abstract Machines. An abstract machine is an implementation schema for an evaluation strategy $\to_{\mathtt{str}}$ with sufficiently atomic operations (accounting for the *machine* part) and without too many details (accounting for the *abstract* part). An abstract machine for $\to_{\mathtt{str}}$ takes care of three tasks:

1. *Search*: searching for $\to_{\mathtt{str}}$-redexes;
2. *Substitution*: replace meta-level substitution with an approximation based on sharing;
3. *Names*: take care of α-equivalence.

These three tasks are left to the *meta-level* in the λ-calculus, meaning that they happen outside the syntax of the calculus itself, in a black-box manner. The reader is most likely acquainted with α-renaming and capture-avoiding substitutions. About search, it is usually specified via a grammar of evaluation contexts, or via deduction rules such as in (1), assuming that, at each application of a rewriting rule, the term is correctly split into an evaluation context and a redex. The meta-level aspect is the fact that the

computational process of splitting the term is not taken into account as an operation of the calculus.

The role of abstract machines is to explicitly take care of these three meta-level aspects, in a possibly efficient way.

Dissecting Abstract Machines. To guide the reader through the different concepts to design and analyze abstract machines, the next two subsections describe in detail two toy machines addressing in isolation the first two mentioned tasks, *search* and *substitution*. They shall then be merged into the Milner Abstract Machine (MAM). We are going to be very careful with names and α-equivalence but we shall not discuss abstract machines that deal with explicitly implementing the third task. We come back to this point at the end of these notes.

Abstract Machines Glossary. First, the basic ingredients of abstract machines.

- An *abstract machine* $\mathtt{M} = (Q, \rightsquigarrow, \cdot^\circ, \underline{\cdot})$ is a transitions system $\rightsquigarrow$ over a set of states, noted Q, where
  - transitions $\rightsquigarrow$ are partitioned into *β-transitions* $\rightsquigarrow_\beta$ and *overhead transitions* $\rightsquigarrow_{\mathtt{o}}$,
  - $\cdot^\circ$ is a a *compilation* function turning λ-terms into states,
  - $\underline{\cdot}$ is a *decoding* function turning states into λ-terms and satisfying the *initialization constraint* $\underline{t^\circ} = t$ for all λ-terms t.
- A state Q is *initial* if $Q = t^\circ$ for some λ-term t, and *final* if no transitions apply.
- An *execution* $r : Q \rightsquigarrow^* Q'$ is a possibly empty sequence of transitions from an initial state to a state Q' said *reachable*.

Now, some more details.

- A state is given by the *code under evaluation* plus some *data-structures* to implement *search* and *substitution*, and to take care of *names*;
- The code under evaluation, as well as the other pieces of code scattered in the data-structures, are λ-terms *not considered modulo α-equivalence*;
- Codes are over-lined, to stress the different treatment of α-equivalence;
- A code $\overline{t}$ is *well-named* if all bound variable names in $\overline{t}$ are distinct and no variable name appears both bound and free;
- The code of initial states is well-named;
- We use $|r|$ for the length of an execution r, and $|r|_\beta$ for the number of β-transitions in r.

Implementations. For every machine one has to prove that it correctly implements the strategy it was conceived for. Our notion, tuned towards complexity analyses, requires a perfect match between the number of β-steps of the strategy and the number of β-transitions of the machine execution.

Definition 2.1 (Machine implementation)**.** *A machine* $\mathtt{M}$ implements a strategy $\to_{\mathtt{str}}$ *on* λ*-terms when given a* λ*-term* t *the following holds*

1. Executions to evaluations*: for any* $\mathtt{M}$*-execution* $r : t^\circ \rightsquigarrow^*_{\mathtt{M}} Q$ *there exists a* $\to_{\mathtt{str}}$*-evaluation* $e : t \to^*_{\mathtt{str}} \underline{Q}$*.*
2. Evaluations to executions*: for every* $\to_{\mathtt{str}}$*-evaluation* $e : t \to^*_{\mathtt{str}} u$ *there exists a* $\mathtt{M}$*-execution* $r : t^\circ \rightsquigarrow^*_{\mathtt{M}} Q$ *such that* $\underline{Q} = u$*.*
3. β-Matching*: in both previous points the number* $|r|_\beta$ *of* β*-transitions in* r *is exactly the length* $|e|$ *of the evaluation* e*, i.e.* $|e| = |r|_\beta$*.*

Note that if a machine implements a strategy than the two are *weakly bisimilar*, where weakness is given by the fact that overhead transitions do not have an equivalent on the calculus (hence their name).

3 The Searching Abstract Machine

Strategies are usually specified through inductive rules as those in (1). The inductive rules incorporate in the definition the search for the next redex to reduce. Abstract machines make such a search explicit and actually ensure two related sub-tasks:

1. Storing the current evaluation context in appropriate *data-structures*.
2. Searching *incrementally*, exploiting previous searches.

For weak CbN reduction the search mechanism is basic. The data structure is simply a stack S storing the arguments of the current head subterm.

Searching Abstract Machine. The searching abstract machine (Searching AM) in Fig. 1 has two components, the *code* in evaluation position and the *argument stack*. The machine has only two transitions, corresponding to the rules in (1), one β-transition ($\rightsquigarrow_\beta$) dealing with β-redexes in evaluation position and one overhead transition ($\rightsquigarrow_{sea}$) adding a term on the argument stack.

Compilation of a (well-named) term t into a machine state simply sends t to the *initial state* $(\overline{t}, \epsilon)$. The decoding given in Fig. 1 is defined inductively on the structure of states. It can equivalently be given contextually, by associating an evaluation context to the data structures—in our case sending the argument stack S to an applicative context $\underline{S}$ by setting $\underline{\epsilon} := \langle\cdot\rangle$, $\underline{\overline{u} :: S} := \underline{S}\langle\langle\cdot\rangle u\rangle$, and then redefining the decoding as $\underline{(\overline{t}, S)} := \underline{S}\langle t\rangle$. It is useful to have both definitions since sometimes one is more convenient than the other. and we will stick to it. A more general and modular approach, however, shall hopefully

$$\begin{array}{rrcl|rrcl} \text{Stacks} & S & := & \epsilon \mid \bar{t} :: S & \text{Decoding} & \underline{(\bar{t}, \epsilon)} & := & t \\ \text{Compilation} & t^{\circ} & := & (\bar{t}, \epsilon) & & \underline{(\bar{t}, \overline{u} :: S)} & := & \underline{(\bar{t}\overline{u}, S)} \end{array}$$

Code	*Stack*	*Trans.*	*Code*	*Stack*
$\bar{t}\overline{u}$	S	$\rightsquigarrow_{sea}$	$\bar{t}$	$\overline{u} :: S$
$\lambda x.\bar{t}$	$\overline{u} :: S$	$\rightsquigarrow_{\beta}$	$\bar{t}\{x \leftarrow \overline{u}\}$	S

Figure 1: Searching Abstract Machine (Searching AM).

be taken into account by future works. A term t should be compiled with respect to an evaluation context E: the term would fill the code component of the machine, as it is standard, and the context should fill up the data-structures, that is not standard.

Implementation. We now show the implementation theorem for the Searching AM with respect to the weak CbN strategy. Despite the simplicity of the machine, we provide a quite accurate account of the proof of the theorem, to be taken as a modular recipe, spelled out in the next section. The proofs of the other implementation theorems in these notes shall then be omitted as they follow exactly the same structure, *mutatis mutandis*.

The *executions-to-evaluations* part of the implementation theorem always rests on a lemma about the decoding of transitions, that in our case takes the following form.

Lemma 3.1 (Transitions Decoding)**.** *Let Q be a Searching AM state.*

1. β-transitions*: if $Q \rightsquigarrow_{\beta} Q'$ then $\underline{Q} \to_{\beta} \underline{Q'}$.*
2. Overhead transitions*: if $Q \rightsquigarrow_{sea} Q'$ then $\underline{Q} = \underline{Q'}$.*

Proof. The first point about β-transitions is more easily proved using the contextual definitions of $\to_{wh}$ and decoding, while the point about overhead transitions follows immediately from the inductive definition of the decoding.

1. $\underline{Q} = \underline{(\lambda x.\bar{t}, \overline{u} :: S)} = \underline{\overline{u} :: S}\langle \lambda x.t \rangle = \underline{S}\langle (\lambda x.t)u \rangle \to_{wh} \underline{S}\langle t\{x \leftarrow u\} \rangle = \underline{Q'}$. Note that the $\to_{wh}$ step can be applied because $\underline{S}$ is an applicative context.
2. $\underline{Q'} = \underline{(\bar{t}, \overline{u} :: S)} = \underline{(\bar{t}\overline{u}, S)} = \underline{Q}$. □

Transitions decoding extends to a projection of executions to evaluations via a straightforward induction on the length of the execution, as required by the implementation theorem. Example:

$$(\lambda x.xx)\mathtt{I}\delta \mid \epsilon \quad \rightsquigarrow_{sea} \quad (\lambda x.xx)\mathtt{I} \mid \delta \quad \rightsquigarrow_{sea} \quad \lambda x.xx \mid \mathtt{I} :: \delta \quad \rightsquigarrow_{\beta} \quad \mathtt{II} \mid \delta$$

decodes to

$$(\lambda x.xx)\mathtt{I}\delta \quad = \quad (\lambda x.xx)\mathtt{I}\delta \quad = \quad (\lambda x.xx)\mathtt{I}\delta \quad \to_{wh} \quad \mathtt{II}\delta$$

For the *evaluations-to-executions* part of the theorem, we proceed similarly, by first proving that single weak CbN steps are simulated by the Searching AM and then extending the simulation to evaluations via an easy induction. There is a subtlety, however, because, if done naively, one-step simulations do not compose.

Let us explain the point. Given a step $t \to_{wh} u$ there exists a state Q such that $t^\circ \rightsquigarrow_{sea}^* \rightsquigarrow_\beta Q$ and $\underline{Q} = u$, as expected. For instance, $(\lambda x.xx)\mathtt{I}\delta \to_{wh} \mathtt{II}\delta$ is simulated by

$$(\lambda x.xx)\mathtt{I}\delta \mid \epsilon \quad \rightsquigarrow_{sea} \quad (\lambda x.xx)\mathtt{I} \mid \delta \quad \rightsquigarrow_{sea} \quad \lambda x.xx \mid \mathtt{I} :: \delta \quad \rightsquigarrow_\beta \quad \mathtt{II} \mid \delta$$

This property, however, cannot be iterated to build a many-steps simulation, because $\underline{Q} = u$ does not imply $Q = u^\circ$, that is, Q in general is not the compilation of u. Extend for instance the last example with a second $\to_{wh}$ step:

$$(\lambda x.xx)\mathtt{I}\delta \to_{wh} \mathtt{II}\delta \to_{wh} \mathtt{I}\delta.$$

We have seen that the simulation of the first steps $(\lambda x.xx)\mathtt{I}\delta \to_{wh} \mathtt{II}\delta$ ends in the machine state $\mathtt{II} \mid \delta$, while the simulation of the second step gives:

$$\mathtt{II}\delta \mid \epsilon \quad \rightsquigarrow_{sea} \quad \mathtt{II} \mid \delta \quad \rightsquigarrow_{sea} \quad \mathtt{I} \mid \mathtt{I} :: \delta \quad \rightsquigarrow_\beta \quad \mathtt{I} \mid \delta$$

Note that $\mathtt{II} \mid \delta \quad \neq \quad \mathtt{II}\delta \mid \epsilon$, that is, the end state of the first execution and the starting state of the second execution do not coincide. Therefore, the two executions cannot be concatenated.

To make things work, the simulation of $t \to_{wh} u$ should not start from t° but from a state Q' such that $\underline{Q'} = t$, that is, the property to iterate should rather be that if $\underline{Q} \to_{wh} u$ then $Q \rightsquigarrow_{sea}^* \rightsquigarrow_\beta Q'$ with $\underline{Q'} = u$ (which is the statement of Lemma 3.3 below). Back to the example, the second steps $\mathtt{II}\delta \to_{wh} \mathtt{I}\delta$ can indeed be seen as $\underline{\mathtt{II} \mid \delta} \to_{wh} \mathtt{I}\delta$ and

$$\mathtt{II} \mid \delta \quad \rightsquigarrow_{sea} \quad \mathtt{I} \mid \mathtt{I} :: \delta \quad \rightsquigarrow_\beta \quad \mathtt{I} \mid \delta$$

Now, we are going to prove the just described step simulation lemma. Its proof relies on the three properties in the statement of the following lemma.

Lemma 3.2 (Properties for step simulation)**.**

1. Overhead transitions terminate*: $\rightsquigarrow_{sea}$ terminates;*
2. Determinism*: the Searching AM is deterministic;*
3. Halt*: final Searching AM states decode to $\to_{wh}$-normal terms.*

Proof. Termination: $\rightsquigarrow_{sea}$-sequences are bound by the size of the code. *Determinism*: $\rightsquigarrow_\beta$ and $\rightsquigarrow_{sea}$ clearly do not overlap and can be applied in a unique way. *Halt*: final states have the form $(\lambda x.\bar{t}, \epsilon)$ and (x, S), that both decode to $\to_{wh}$-normal forms. □

Lemma 3.3 (One-step simulation)**.** *Let Q be a Searching AM state. If $\underline{Q} \to_{wh} u$ then there exists a state Q' such that $Q \rightsquigarrow_{sea}^* \rightsquigarrow_\beta Q'$ and $\underline{Q'} = u$.*

Proof. Let $\mathtt{nf}_{sea}(Q)$ be the normal form of Q with respect to $\rightsquigarrow_{sea}$, that exists and is unique by termination of $\rightsquigarrow_{sea}$ (Lemma 3.2.1) and determinism of the machine (Lemma 3.2.2). Since $\rightsquigarrow_{sea}$ is mapped on identities (Lemma 3.1.2) one has $\underline{\mathtt{nf}_{sea}(Q)} = \underline{Q}$. By hypothesis $\underline{Q}$ $\rightarrow_{wh}$-reduces, so that by the halt property (Lemma 3.2.3) $\mathtt{nf}_{sea}(Q)$ cannot be final. Then $\mathtt{nf}_{sea}(Q) \rightsquigarrow_{\beta} Q'$, and $\underline{\mathtt{nf}_{sea}(Q)} = \underline{Q} \rightarrow_{wh} \underline{Q'}$ by the transitions decoding lemma (Lemma 3.1.1). By determinism of $\rightarrow_{wh}$, one obtains $\underline{Q'} = u$. □

Finally, we obtain the implementation theorem.

Theorem 3.4. *The Searching AM implements the weak CbN strategy.*

Proof. *Executions to evaluations*: by induction on the length $|r|$ of r using Lemma 3.1. *evaluations to Executions*: by induction on the length $|e|$ of e using Lemma 3.3 and noting that $\underline{t^{\circ}} = t$. □

4 Abstract Implementations

The proof of the implementation theorem given above can be abstracted to that of a generic machine $\mathtt{M}$ with transitions $\rightsquigarrow_{\mathtt{M}}$ (defined as the union of β-transitions $\rightsquigarrow_{\beta}$ and overhead transitions $\rightsquigarrow_{\mathtt{o}}$) implementing a strategy $\rightarrow_{\mathtt{str}}$, as we now show. This is useful to have a recipe for implementation theorems for other machines.

First, we abstract the properties used in the previous section.

Definition 4.1 (Implementation system)**.** *A machine* $\mathtt{M}$*, a strategy* $\rightarrow_{\mathtt{str}}$*, and a decoding* $\underline{\cdot}$ *form an* implementation system *if the following conditions hold:*

1. β-projection*:* $Q \rightsquigarrow_{\beta} Q'$ *implies* $\underline{Q} \rightarrow_{\mathtt{str}} \underline{Q'}$*;*
2. Overhead transparency*:* $Q \rightsquigarrow_{\mathtt{o}} Q'$ *implies* $\underline{Q} = \underline{Q'}$*;*
3. Overhead transitions terminate*:* $\rightsquigarrow_{\mathtt{o}}$ *terminates;*
4. Determinism*:* $\rightarrow_{\mathtt{str}}$ *is deterministic; [I changed this requirement, while working on the paper with Pablo. I changed the proof below, but I have not checked the rest of the document]*
5. Halt*:* $\mathtt{M}$ *final states decode to* $\rightarrow_{\mathtt{str}}$*-normal terms.*

Then, we abstract the one-step simulation lemma (Lemma 3.3).

Lemma 4.2 (One-step simulation)**.** *Let* $\mathtt{M}$*,* $\rightarrow_{\mathtt{str}}$*, and* $\underline{\cdot}$ *be a machine, a strategy, and a decoding forming an implementation system. For any state* Q *of* $\mathtt{M}$*, if* $\underline{Q} \rightarrow_{\mathtt{str}} u$ *then there is a state* Q' *of* $\mathtt{M}$ *such that* $Q \rightsquigarrow_{\mathtt{o}}^{*} \rightsquigarrow_{\beta} Q'$ *and* $\underline{Q'} = u$*.*

Proof. For any state Q of $\mathtt{M}$, let $\mathtt{nf}_{\mathtt{o}}(Q)$ a normal form of Q with respect to $\rightsquigarrow_{\mathtt{o}}$: such a state exists unique because overhead transitions terminate (Point 3). Since $\rightsquigarrow_{\mathtt{o}}$ is mapped on identities (Point 2), one has $\underline{\mathtt{nf}_{\mathtt{o}}(Q)} = \underline{Q}$. As $\underline{Q}$ is not $\rightarrow_{\mathtt{str}}$-normal by hypothesis, the halt property (Point 5) entails that $\mathtt{nf}_{\mathtt{o}}(Q)$ is not final, therefore $Q \rightsquigarrow_{\mathtt{o}}^{*} \mathtt{nf}_{\mathtt{o}}(Q) \rightsquigarrow_{\beta} Q'$ for some state Q', and thus $\underline{Q} = \underline{\mathtt{nf}_{\mathtt{o}}(Q)} \rightarrow_{\mathtt{str}} \underline{Q'}$ by β-projection (Point 1). By determinism of $\rightarrow_{\mathtt{str}}$ (Point 4), one obtains $\underline{Q'} = u$. □

Finally, we abstract the implementation theorem.

Theorem 4.3 (Sufficient condition for implementations)**.** *Let* $(\mathtt{M}, \to_{\mathtt{str}}, \underline{\cdot})$ *be an* implementation system*. Then,* $\mathtt{M}$ *implements* $\to_{\mathtt{str}}$ *via* $\underline{\cdot}$*.*

Proof. According to Definition 2.1, given a λ-term t, we have to show that:

1. *Executions to evaluations with β-matching*: for any $\mathtt{M}$-execution $r : t^\circ \rightsquigarrow^*_{\mathtt{M}} Q$ there exists a $\to_{\mathtt{str}}$-evaluation $e : t \to^*_{\mathtt{str}} \underline{Q}$ such that $|e| = |r|_\beta$.
2. *Evaluations to executions with β-matching*: for every $\to_{\mathtt{str}}$-evaluation $e : t \to^*_{\mathtt{str}} u$ there exists a $\mathtt{M}$-execution $r : t^\circ \rightsquigarrow^*_{\mathtt{M}} Q$ such that $\underline{Q} = u$ and $|e| = |r|_\beta$.

Proof of Point 1 By induction on $|r|_\beta \in \mathbb{N}$.

If $|r|_\beta = 0$ then $r\colon t^\circ \rightsquigarrow^*_{\mathtt{o}} Q$ and hence $\underline{t^\circ} = \underline{Q}$ by overhead transparency (Point 2 of Definition 4.1). Moreover, $t = \underline{t^\circ}$ since decoding is the inverse of compilation on initial states, therefore we are done by taking the empty (i.e. without steps) evaluation e with starting (and end) term t.

Suppose $|r|_\beta > 0$: then, $r\colon t^\circ \rightsquigarrow^*_{\mathtt{M}} Q$ is the concatenation of an execution $r'\colon t^\circ \rightsquigarrow^*_{\mathtt{M}} Q'$ followed by an execution $r''\colon Q' \rightsquigarrow_\beta Q'' \rightsquigarrow^*_{\mathtt{o}} Q$. By *i.h.* applied to r', there exists an evaluation $e'\colon t \to^*_{\mathtt{str}} \underline{Q'}$ with $|r'|_\beta = |e'|$. By β-projection (Point 1 of Definition 4.1) and overhead transparency (Point 2 of Definition 4.1) applied to r'', one has $e''\colon \underline{Q'} \to_{\mathtt{str}} \underline{Q''} = \underline{Q}$. Therefore, the evaluation e defined as the concatenation of e' and e'' is such that $e\colon t \to^*_{\mathtt{str}} \underline{Q}$ and $|e| = |e'| + |e''| = |r'|_\beta + 1 = |r|_\beta$.

Proof of Point 2 By induction on $|e| \in \mathbb{N}$.

If $|e| = 0$ then $t = u$. Since decoding is the inverse of compilation on initial states, one has $\underline{t^\circ} = t$. We are done by taking the empty (i.e. without transitions) execution r with initial (and final) state t°.

Suppose $|e| > 0$: so, $e\colon t \to^*_{\mathtt{str}} u$ is the concatenation of an evaluation $e'\colon t \to^*_{\mathtt{str}} u'$ followed by the step $u' \to_{\mathtt{str}} u$. By *i.h.*, there exists a $\mathtt{M}$-execution $r'\colon t^\circ \rightsquigarrow^*_{\mathtt{M}} Q'$ such that $\underline{Q'} = u'$ and $|e'| = |r'|_\beta$. By one-step simulation (Lemma 4.2, since $\underline{Q'} \to_{\mathtt{str}} u$ and $(\mathtt{M}, \to_{\mathtt{str}}, \underline{\cdot})$ is an implementation system), there is a state Q of $\mathtt{M}$ such that $Q' \rightsquigarrow^*_{\mathtt{o}} \rightsquigarrow_\beta Q$ and $\underline{Q} = u$. Therefore, the execution $r\colon t^\circ \rightsquigarrow^*_{\mathtt{M}} Q' \rightsquigarrow^*_{\mathtt{o}} \rightsquigarrow_\beta Q$ is such that $|r|_\beta = |r'|_\beta + 1 = |e'| + 1 = |e|$. □

5 The CbN Micro-Substituting Abstract Machine

Decomposing Meta-Level Substitution. The second task of abstract machines is to replace meta-level substitution $\overline{t}\{x \leftarrow \overline{u}\}$ with *micro-step substitution on demand*, i.e. a parsimonious approximation of meta-level substitution based on:

1. *Sharing*: when a β-redex $(\lambda x.\overline{t})\overline{u}$ is in evaluation position it is fired but the meta-level substitution $\overline{t}\{x \leftarrow \overline{u}\}$ is delayed, by introducing an annotation $[x \leftarrow \overline{u}]$—that is, an explicit substitution—in a data-structure for delayed substitutions called *environment*;

2. *Micro-step substitution*: variable occurrences are replaced one at a time;

3. *Substitution on demand*: replacement of a variable occurrence happens only when it ends up in evaluation position—variable occurrences that do not end in evaluation position are never substituted.

The purpose of this section is to illustrate this process in isolation via the study of a toy machine, the *CbN Micro-Substituting Abstract Machine* (CbN Micro AM) in Fig. 2, forgetting about the search for redexes.

The CbN Micro AM is in fact just a minor variation over the micro weak head reduction of the previous lecture, where the explicit substitutions are collected together in an environment rather than scattered through the term structure. This is typical of abstract machines.

Environments. An *environment* E is a list of entries of the form $[x \leftarrow \overline{u}]$. Each entry denotes the *delayed* substitution of $\overline{u}$ for x. There is a key property of environments that, as it is stated by forthcoming Lemma 5.1, holds for every rechable state $(\overline{t}, E' :: [x \leftarrow \overline{u}] :: E'')$:

> *Scope*: the scope of x is given by $\overline{t}$ and E', that is, x is fresh with respect to $\overline{u}$ and E''.

A consequence is that, for the environment $[x_1 \leftarrow \overline{u}_1] \dots [x_k \leftarrow \overline{u}_k]$ of a reachable state, the variables $x_1, \dots, x_k$ are all distinct.

The (global) environment models a store. As it is standard in the literature, it is a *list*, but the list structure is only used to obtain a simple decoding and a handy delimitation of the scope of its entries. These properties are useful to develop the meta-theory of abstract machines, but keep in mind that (global) environments are not meant to be implemented as lists.

Code. The code under evaluation is now a λ-term $h\overline{s}_1 \dots \overline{s}_k$ expressed as a head h (that is either a β-redex $(\lambda x.\overline{t})\overline{u}$ or a variable x) applied to $k \geq 0$ arguments—it is a by-product of the fact that the CbN Micro AM does not address *search*. Note that, by exploiting the notion of applicative context, the CbN Micro AM can be reformulated as follows:

$Code$	Env	$Trans$	$Code$	Env
$A\langle(\lambda x.\overline{t})\overline{u}\rangle$	E	$\rightsquigarrow_\beta$	$A\langle\overline{t}\rangle$	$[x \leftarrow \overline{u}] :: E$
$A\langle x\rangle$	$E :: [x \leftarrow \overline{t}] :: E'$	$\rightsquigarrow_{sub}$	$A\langle\overline{t}^{\alpha}\rangle$	$E :: [x \leftarrow \overline{t}] :: E'$

Transitions. There are two transitions:

- *Delaying* β: transition $\rightsquigarrow_\beta$ removes the β-redex $(\lambda x.\overline{t})\overline{u}$ but does not execute the expected substitution $\{x \leftarrow \overline{u}\}$, it rather delays it, adding $[x \leftarrow \overline{u}]$ to the environment. It is the β-transition of the CbN Micro AM.

$$\begin{array}{rrcl|rrcl} \text{Environments} & E & := & \epsilon \mid [x{\leftarrow}\overline{t}] :: E & \text{Decoding} & \underline{(\overline{t}, \epsilon)} & := & t \\ \text{Compilation} & t^{\circ} & := & (\overline{t}, \epsilon) & & \underline{(\overline{t}, [x{\leftarrow}\overline{u}] :: E)} & := & \underline{(\overline{t}\{x{\leftarrow}\overline{u}\}, E)} \end{array}$$

Code	*Env*	*Trans*	*Code*	*Env*
$(\lambda x.\overline{t})\overline{u}\overline{s}_1 \dots \overline{s}_k$	E	$\rightsquigarrow_{\beta}$	$\overline{t}\overline{s}_1 \dots \overline{s}_k$	$[x{\leftarrow}\overline{u}] :: E$
$x\overline{s}_1 \dots \overline{s}_k$	$E :: [x{\leftarrow}\overline{t}] :: E'$	$\rightsquigarrow_{sub}$	$\overline{t}^{\alpha}\overline{s}_1 \dots \overline{s}_k$	$E :: [x{\leftarrow}\overline{t}] :: E'$

where $\overline{t}^{\alpha}$ denotes a well-named copy of $\overline{t}$ where bound names have been freshly renamed.

Figure 2: CbN Micro-substituting Abstract Machine (CbN Micro AM).

- *Micro-substitution on demand*: if the head of the code is a variable x and there is an entry $[x{\leftarrow}\overline{t}]$ in the environment then transition $\rightsquigarrow_{sub}$ replaces that occurrence of x—and only that occurrence—with a copy of $\overline{t}$. It is necessary to rename the new copy of $\overline{t}$ (into a well-named term) to avoid name clashes. It is the overhead transition of the CbN Micro AM. Assuming the *scope* property of environments (proved below for reachable states), there is at most one entry for x in the environment, and so the transition is deterministic.

Implementation. Compilation sends a (well-named) term t to the initial state $(\overline{t}, \epsilon)$, as for the Searching AM (but now the empty data-structure is the environment). The decoding (defined in Fig. 2) simply applies the delayed substitutions in the environment to the term, considering them as meta-level substitutions.

The implementation of weak CbN reduction $\rightarrow_{wh}$ by the CbN Micro AM can be shown using the recipe given for the Searching AM. The only relevant difference is in the proof that the overhead transition $\rightsquigarrow_{sub}$ terminates, that is based on a different argument. We spell it out because it shall be useful also later on for complexity analyses. It requires the following invariant of machine executions:

Lemma 5.1 (Name invariant)**.** *Let $Q = (\overline{t}, E)$ be a CbN Micro AM reachable state.*

1. Abstractions*: if $\lambda x.\overline{u}$ is a subterm of $\overline{t}$ or of any code in E then x may occur only in $\overline{u}$, and only free, if at all;*
2. Environment scope*: if $E = E' :: [x{\leftarrow}\overline{u}] :: E''$ then x is fresh with respect to $\overline{u}$ and E'';*

Proof. By induction on the length of the execution r leading to Q. If r is empty then Q is initial and the statement holds because $\overline{t}$ is well-named by hypothesis and the environment is empty. If r is non-empty then let $Q' \rightsquigarrow Q$ its last transition. The statement follows easily from the *i.h.* and the fact that transitions preserve the invariant, but it is instructive to spell out the proof. Cases of $Q' \rightsquigarrow Q$ (referring to the notation of Fig. 2):

- *β transition $\rightsquigarrow_\beta$*: Point 1 follows from Point 1 of the *i.h.* Point 2 for the new environment entry $[x{\leftarrow}\overline{u}]$ also follows from Point 1 of the *i.h.* For the other entries, it follows from Point 2 of the *i.h.*

- *Substitution transition $\rightsquigarrow_{sub}$*: Point 1 follows from Point 1 of the *i.h.* (for $\overline{s}_1 \dots \overline{s}_k$ and the environments E and E'). For the head $\overline{t}^{\alpha}$ of the code and for $\overline{t}$ in the environment it is given by the fact that $\overline{t}^{\alpha}$ is a *well-named* and freshly renamed copy of $\overline{t}$ from the environment by definition of the transition. Point 2 follows from Point 2 of the *i.h.* □

We write $|E|$ for the number of explicit substitutions in E.

Lemma 5.2 (Micro-substitution terminates)**.** *$\rightsquigarrow_{sub}$ terminates in at most $|E|$ steps (on reachable states).*

Proof. Consider a $\rightsquigarrow_{sub}$ transition copying $\overline{u}$ from the environment $E = E' :: [x{\leftarrow}\overline{u}] :: E''$. If the next transition is again $\rightsquigarrow_{sub}$, then the head of $\overline{u}$ is a variable y and the transition copies from an entry in E'' because by Lemma 5.1 y cannot be bound by the entries in E'. Then the number of consecutive $\rightsquigarrow_{sub}$ transitions is bound by $|E|$ (that is not extended by $\rightsquigarrow_{sub}$). □

Theorem 5.3. *The CbN Micro AM and the weak CbN strategy form an implementation system, that is, the following conditions hold:*

1. β-projection*: $Q \rightsquigarrow_\beta Q'$ implies $\underline{Q} \to_{wh} \underline{Q'}$;*
2. Overhead transparency*: $Q \rightsquigarrow_{sub} Q'$ implies $\underline{Q} = \underline{Q'}$;*
3. Overhead transitions terminate*: $\rightsquigarrow_{sub}$ terminates;*
4. Determinism*: the CbN Micro AM is deterministic;*
5. Halt*: CbN Micro AM final states decode to $\to_{wh}$-normal terms.*

Proof.

1. By induction on the definition of the decoding. Two cases:
   - *Base case*: if $Q = (A\langle(\lambda x.\overline{t})\overline{u}\rangle, \epsilon) \rightsquigarrow_\beta (A\langle\overline{t}\rangle, [x{\leftarrow}\overline{u}]) = Q'$ then $\underline{Q} = A\langle(\lambda x.t)u\rangle \to_{wh} A\langle t\{x{\leftarrow}u\}\rangle$. By the abstraction part of the name invariant (Lemma 5.1), x occurs only in t and not in A, so that $A\langle t\{x{\leftarrow}u\}\rangle = A\langle t\rangle\{x{\leftarrow}u\} = \underline{Q'}$.
   - *Inductive case*: if $Q = (A\langle(\lambda x.\overline{t})\overline{u}\rangle, [y{\leftarrow}\overline{s}]E) \rightsquigarrow_\beta (A\langle\overline{t}\rangle, [x{\leftarrow}\overline{u}][y{\leftarrow}\overline{s}]E) = Q'$ then $\underline{Q} = \underline{(A\{y{\leftarrow}\overline{s}\}\langle(\lambda x.\overline{t}\{y{\leftarrow}\overline{s}\})\overline{u}\{y{\leftarrow}\overline{s}\}\rangle, E)}$. It is easily seen that applicative contexts are stable by substitution, so that we have

     $$Q_1 := A\{y{\leftarrow}\overline{s}\}\langle(\lambda x.\overline{t}\{y{\leftarrow}\overline{s}\})\overline{u}\{y{\leftarrow}\overline{s}\}\rangle \mid E \quad \rightsquigarrow_\beta \quad A\{y{\leftarrow}\overline{s}\}\langle\overline{t}\{y{\leftarrow}\overline{s}\}\rangle \mid [x{\leftarrow}\overline{u}\{y{\leftarrow}\overline{s}\}]E =: Q'_1$$

     By *i.h.*, $\underline{Q_1} \rightsquigarrow_\beta \underline{Q'_1}$. Now,

     $$\underline{Q'_1} = \underline{A\{y{\leftarrow}\overline{s}\}\{x{\leftarrow}\overline{u}\{y{\leftarrow}\overline{s}\}\}\langle\overline{t}\{y{\leftarrow}\overline{s}\}\{x{\leftarrow}\overline{u}\{y{\leftarrow}\overline{s}\}\}\rangle \mid E}$$

By the name invariant, x cannot occur in $\overline{u}$. Then by basic properties of substitutions, $\{y{\leftarrow}\overline{s}\}\{x{\leftarrow}\overline{u}\{y{\leftarrow}\overline{s}\}\} = \{x{\leftarrow}\overline{u}\}\{y{\leftarrow}\overline{s}\}$, so that

$$\underline{Q'_1} = \underline{A\{x{\leftarrow}\overline{u}\}\{y{\leftarrow}\overline{s}\}\langle \overline{t}\{x{\leftarrow}\overline{u}\}\{y{\leftarrow}\overline{s}\}\rangle \mid E} = \underline{Q'}$$

2. The decoding can be seen as the application of a substitution σ_E induced by the environment over the code. Consider the transition $Q = A\langle x\rangle \mid E[x{\leftarrow}\overline{t}]E' \rightsquigarrow_{sub} A\langle \overline{t}^{\alpha}\rangle \mid E[x{\leftarrow}\overline{t}]E' = Q'$. We have $\underline{Q} = A\langle x\rangle\sigma_E\{x{\leftarrow}t\}\sigma_{E'}$ and $\underline{Q'} = A\langle \overline{t}^{\alpha}\rangle\sigma_E\{x{\leftarrow}t\}\sigma_{E'}$. By the name invariant, the variables bound by E cannot appear in $\overline{t}$, and neither can x. Then, since on the calculus we work up to α-renaming, we have $A\langle \overline{t}^{\alpha}\rangle\sigma_E\{x{\leftarrow}t\} = A\sigma_E\langle \overline{t}^{\alpha}\rangle\{x{\leftarrow}t\} = A\sigma_E\langle x\rangle\{x{\leftarrow}t\} = A\langle x\rangle\sigma_E\{x{\leftarrow}t\}$, from which $\underline{Q} = \underline{Q'}$ immediately follows.

3. This is Lemma 5.2.

4. By the environment part of the name invariant (Lemma 5.1), there cannot be two entries of the environment bounding the same variable. Therefore, $\rightsquigarrow_{sub}$ is deterministic, and so is the CbN Micro AM.

5. The machine is stuck, and thus the state is final, in two cases:
   - the term is an abstraction with no arguments, that is, $Q = (\lambda x.t, E)$. Then Q decodes to an abstraction, which is a $\rightarrow_{wh}$ normal form.
   - the weak head variable is free, that is, it has no corresponding entry in the environment. Formally, $Q = (x\overline{s}_1 \dots \overline{s}_k, E)$ and $x \notin \mathtt{dom}(E)$. Then Q decodes to $xp_1 \dots p_k$ for some $p_1 \dots p_k$, which is a $\rightarrow_{wh}$ normal form. □

Corollary 5.4. *The CbN Micro AM implements the weak CbN strategy $\rightarrow_{wh}$.*

Proof. By Theorem 4.3 and Theorem 5.3. □

Name Clashes. Is the renaming $\overline{t}^{\alpha}$ done by the substitution transition $\rightsquigarrow_{sub}$ really needed? Yes, it is. Let us consider the Non-Renaming CbN Micro AM, that is exactly as the CbN Micro AM except that its transition $\rightsquigarrow_{sub}$ simply substitutes $\overline{t}$ instead of $\overline{t}^{\alpha}$. The environment scope invariant no longer holds, as its proof relies on the renaming. Therefore, in the Non-Renaming CbN Micro AM there may be many environment entries bounding the same variable x, and so $\rightsquigarrow_{sub}$ becomes non-deterministic. Of course, one can decide to always substitute the code $\overline{u}$ associated to the *leftmost* entry $[x{\leftarrow}\overline{u}]$ in the environment, for instance. But then try to evaluate with the Non-Renaming CbN Micro AM the term $t = (\lambda x.xx(\lambda y.y)(\lambda z.zz))(\lambda w.\lambda k.wk)$. The evaluation in the λ-calculus of t produces the normal form $\lambda z.zz$. What happens when instead one executes the Non-Renaming CbN Micro AM on t?

$$
\begin{array}{rrcl}
\text{Environments} & E & := & \epsilon \mid [x{\leftarrow}\overline{t}] :: E \\
\text{Stacks} & S & := & \epsilon \mid \overline{t} :: S \\
\text{Compilation} & t^{\circ} & := & (\overline{t}, \epsilon, \epsilon)
\end{array}
\qquad
\begin{array}{rrcl}
\text{Decoding} & \underline{(\overline{t}, \epsilon, \epsilon)} & := & t \\
& \underline{(\overline{t}, \overline{u} :: S, E)} & := & \underline{(\overline{t}\overline{u}, S, E)} \\
& \underline{(\overline{t}, \epsilon, [x{\leftarrow}\overline{u}] :: E)} & := & \underline{(\overline{t}\{x{\leftarrow}\overline{u}\}, \epsilon, E)}
\end{array}
$$

Code	*Stack*	*Env*	*Trans*	*Code*	*Stack*	*Env*
$\overline{t}\overline{u}$	S	E	$\rightsquigarrow_{sea}$	$\overline{t}$	$\overline{u} :: S$	E
$\lambda x.\overline{t}$	$\overline{u} :: S$	E	$\rightsquigarrow_{\beta}$	$\overline{t}$	S	$[x{\leftarrow}\overline{u}] :: E$
x	S	$E :: [x{\leftarrow}\overline{t}] :: E'$	$\rightsquigarrow_{sub}$	$\overline{t}^{\alpha}$	S	$E :: [x{\leftarrow}\overline{t}] :: E'$

where $\overline{t}^{\alpha}$ denotes a well-named copy of $\overline{t}$ where bound names have been freshly renamed.

Figure 3: Milner Abstract Machine (MAM).

6 Search + Micro-Substitution = Milner Abstract Machine

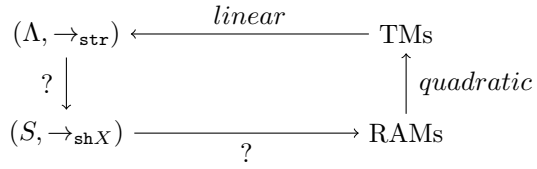
The Searching AM and the CbN Micro AM can be merged together into the Milner Abstract Machine (MAM), defined in Fig. 3. The MAM has both an argument stack and an environment. The machine has one β-transition $\rightsquigarrow_{\beta}$ inherited from the Searching AM, and two overhead transitions, $\rightsquigarrow_{sea}$ inherited from the the Searching AM and $\rightsquigarrow_{sub}$ inherited from the CbN Micro AM. Note that in $\rightsquigarrow_{sub}$ the code now is simply a variable, because the arguments are supposed to be stored in the argument stack. Compilation sends a term t to the initial state $(\overline{t}, \epsilon, \epsilon)$ and decoding (in Fig. 3) first restores the codes in the stack as argument of the main code, and then turns the environment into meta-level substitutions, as expected. Of course, the decoding can equivalently be done the other way around, by first decoding the environment and then the stack, the two operations commute.

For the implementation theorem, once again the delicate point is to prove that the overhead transitions terminate. As for the CbN Micro AM one needs a name invariant. A termination measure can then be defined easily by mixing the size of the codes (needed for $\rightsquigarrow_{sea}$) and the size of the environment (needed for $\rightsquigarrow_{sub}$), and it is omitted here, because it shall be studied exhaustively for the complexity analysis of the MAM. By reasoning along the same lines as for the CbN Micro AM, we obtain that:

Theorem 6.1. *The MAM implements the weak CbN strategy.*

7 Introducing Complexity Analyses

The complexity analysis of abstract machines is the study of the asymptotic behavior of their overhead. Let's go back to the initial diagram:



The complexity of the abstract machine is the overhead of the composition of the left and the bottom arrows. We refer to the left and bottom arrows as to the high-level and low-level simulations, respectively. We first deal with the complexity of the low-level simulation, which is somewhat simpler, and then with the high-level one.

7.1 Complexity of the Low-Level Simulation

The Sub-Term Property. For the moment, we discuss a generic, unspecified machine. The overhead of the low-level simulation is usually linear. More precisely, given a machine execution $r : t_0^\circ \rightsquigarrow^n Q$ the cost of implementing r on RAM is *bi-linear*, that is, linear in:

- *Input*: the size $|t_0|$ of the initial term;
- *Low-level time*: the number n of transitions.

The bi-linear bound is always obtained via the *sub-term property* of the machine. In the case of the MAM it takes the following form (the MAM never erases, which is why the property does not say anything about erased codes).

Lemma 7.1.

1. MAM sub-term invariant*: if $r : t_0^\circ \rightsquigarrow_{MAM} (\overline{u}, S, E)$ is a MAM execution then $\overline{u}$ and any code in S and E are sub-terms of t_0 up to variables renaming.*
2. MAM sub-term property*: all codes duplicated by the MAM have size $\leq |t_0|$.*

Proof. Point 1 is by induction on the length of r. For the base case the statement trivially holds. For the inductive case, one analyzes every transitions and the invariant immediately follows from the *i.h.*.

Point 2 is a direct consequence of point 1, since the MAM duplicates codes only in transition $\rightsquigarrow_{sub}$ and taking them from the environment, which contains sub-terms of the initial code up to renamings. □

As we have seen in the previous lecture, the sub-term property forbids size explosion. But be careful: here it forbids size explosion with respect to the number of machine transitions, which for the moment is not related to the number of β-steps. Moreover, the property usually implies the *reasonable steps property*, and the fact that the MAM can be simulated with polynomial overhead by a RAM. We can however be more precise.

Cost of Single Transitions. To provide precise bounds on the cost of implementing the MAM, we need to make some hypotheses on the implementation, since the abstract specification of the machine is too vague:

1. *Codes, variable (occurrences), and environment entries*: abstractions and applications are constructors with pointers to sub-terms, a variable is a memory location, a variable occurrence is a reference to that location, and an environment entry $[x \leftarrow \bar{t}]$ is the fact that the location associated to x contains (the topmost constructor of) $\bar{t}$.

2. *Random access to global environments*: accessing the environment E in transition $\rightsquigarrow_{sub}$ can be done in $\mathcal{O}(1)$ by just following the reference given by the variable occurrence under evaluation, with no need to access E sequentially, thus ignoring its list structure.

3. *Linear time renaming*: the renaming operation $\bar{t}^{\alpha}$ in transition $\rightsquigarrow_{sub}$ can be done in time $\mathcal{O}(|\bar{t}|)$.

These hypotheses are realistic. At the end of this lecture we discuss an OCaml implementation realizing these hypotheses taken from a paper by Accattoli and Barras [AB17], where alternative machines and implementations are also discussed.

Let us point out that the RAM model is used informally as the model behind the usual way of measuring the complexity of algorithms in pseudo-code, and that the formal specification on a RAM is never actually carried out.

It is now possible to bound the cost of single transitions, from which the cost of executions follows. Note that the case of $\rightsquigarrow_{sub}$ transitions relies on the sub-term invariant.

Proposition 7.2 (Cost of single transitions, reasonable steps for the MAM)**.** *Let $r : t_0^{\circ} \rightsquigarrow_{MAM}^{*} Q$ be a MAM execution. Then:*

1. *Each $\rightsquigarrow_{sea}$ transition in r is implemented in $\mathcal{O}(1)$ time on RAM;*

2. *Each $\rightsquigarrow_{\beta}$ transition in r is implemented in $\mathcal{O}(1)$ time on RAM;*

3. *Each $\rightsquigarrow_{sub}$ transition in r is implemented in $\mathcal{O}(|t_0|)$ time on RAM.*

Therefore, the MAM has reasonable steps.

Proof. According to our hypothesis on the concrete implementation of the MAM, $\rightsquigarrow_{sea}$ just moves the pointer to the current code on the left sub-term of the application and pushes the pointer to the right sub-term on the stack—evidently constant time. Similarly for $\rightsquigarrow_{\beta}$. For $\rightsquigarrow_{sub}$, the environment entry $[x \leftarrow \bar{t}]$ is accessed in constant time by hypothesis. Then $\bar{t}$ has to be α-renamed, which by hypothesis is done in time $\mathcal{O}(|\bar{t}|)$ where $\bar{t}$ is the duplicated code. By the sub-term property (Lemma 7.1), $|\bar{t}| \leq |t_0|$, thus the bound is $\mathcal{O}(|t_0|)$. □

Corollary 7.3 (The low-linear simulation is bi-linear)**.** *Let $r : t_0^{\circ} \rightsquigarrow_{MAM}^{n} Q$ be a MAM execution. Then r can by implemented on RAM in time $\mathcal{O}(n \cdot |t_0|)$.*

Proof. Straightforward induction on n, using Proposition 7.2 in the inductive case. □

8 Complexity of the High-Level Simulation

The complexity analysis of the high-level simulation, that is, of the simulation of weak head reduction by the MAM, requires to estimate how many machine transitions there can be in the simulation of a reduction sequence.

Parameters for Complexity Analyses. Let us reason abstractly, by considering a generic strategy $\to_{\mathtt{str}}$ in the λ-calculus and a given machine $\mathtt{M}$ implementing $\to_{\mathtt{str}}$. By the *evaluations-to-executions* part of the implementation theorem (Definition 2.1), given an evaluation $e : t_0 \to^n_{\mathtt{str}} u$ there is a shortest execution $r : t_0^\circ \rightsquigarrow_{\mathtt{M}} Q$ such that $\underline{Q} = u$. Determining *the complexity of the high-level simulation* amounts to bound the length of r depending as a function of two fundamental parameters:

1. *Input*: the size $|t_0|$ of the initial term t_0 of the evaluation e;
2. *Abstract time*: the length $n = |e|$ of the evaluation e, that coincides with the number $|r|_\beta$ of β-transitions in r by the β-matching requirement for implementations.

Note that our notion of implementation allows us to forget about the strategy while studying the complexity of the machine, because the two fundamental parameters are internalized: the input is simply the initial code and the length of the strategy is simply the number of β-transitions.

Analysing Each Transition. In order to compose the analyses of the high-level and low-level simulations, we need to know a bit more than the length of the machine execution. We need to know how many transitions there are in an execution for each kind of transitions, as to then multiply each kind for its the cost, and sum over transition kinds. For the MAM, we shall then bound the number of substitution transitions and the number of search transitions separately.

Types of Machines. The bound obtained by composing the high-level and low-level analyses is then used to classify the machine, as follows.

Definition 8.1. *Let* $\mathtt{M}$ *be an abstract machine implementing a strategy* $\to_{\mathtt{str}}$*. Then*

- $\mathtt{M}$ *is* reasonable *if the complexity of* $\mathtt{M}$ *is polynomial in the input* $|t_0|$ *and its abstract time* $|r|_\beta$*;*
- $\mathtt{M}$ *is* unreasonable *if it is not reasonable;*
- $\mathtt{M}$ *is* efficient *if it is bi-linear, that is, linear in both the input and its abstract time.*

Note a potential source of confusion: proving that a strategy provides an unreasonable cost model requires to show that *all* its simulations necessarily have a more than polynomial overhead, while a machine is unreasonable when the single simulation that it provides has a more than polynomial overhead.

8.1 A Partial High-Level Analysis of the CbN Micro AM

Here we bound the number of overhead transition for the CbN Micro AM, in order to factor the reasoning needed for the MAM and introduce some concepts in a simpler setting.

Number of substitution transitions. The next lemma bounds the global number of overhead transitions. It relies on an auxiliary bound of a more local form that is a direct consequence of the termination of substitution transitions (Lemma 5.2). We use $|r|_{sub}$ for the number of substitution transitions in r.

Lemma 8.2. *Let $r : t_0^\circ \rightsquigarrow_{MAM} Q$ be a CbN Micro AM execution.*

1. Micro-substitution linear local bound*: if $r' : Q \rightsquigarrow_{sub}^* Q'$ then $|r'|_{sub} \leq |E| = |r|_\beta$;*
2. Micro-substitution quadratic global bound*: $|r|_{sub} \leq |r|_\beta^2$.*

Proof.

1. By Lemma 5.2, $|r'|_{sub} \leq |E|$. Now, $|E|$ is exactly $|r|_\beta$, because the only transition extending E, and of exactly one entry, is $\rightsquigarrow_\beta$.
2. The fact that a linear local bound induces a quadratic global bound is a standard reasoning. We spell it out to help the unacquainted reader. The execution r alternates phases of β-transitions and phases of overhead transitions, i.e. it has the shape:

$$t_0^\circ = Q_1 \rightsquigarrow_\beta^* Q'_1 \rightsquigarrow_{sub}^* Q_2 \rightsquigarrow_\beta^* Q'_2 \rightsquigarrow_{sub}^* \dots Q_k \rightsquigarrow_\beta^* Q'_k \rightsquigarrow_{sub}^* Q$$

   Let a_i be the length of the segment $Q_i \rightsquigarrow_\beta^* Q'_i$ and b_i be the length of the segment $Q'_i \rightsquigarrow_{sub}^* Q_{i+1}$, for $i = 1, \dots, k$. By Point 1, we obtain $b_i \leq \sum_{j=1}^i a_j$. Then $|r|_{sub} = \sum_{i=1}^k b_i \leq \sum_{i=1}^k \sum_{j=1}^i a_j$. Note that $\sum_{j=1}^i a_j \leq \sum_{j=1}^k a_j = |r|_\beta$ and $k \leq |r|_\beta$. So $|r|_{sub} \leq \sum_{i=1}^k \sum_{j=1}^i a_j \leq \sum_{i=1}^k |r|_\beta \leq |r|_\beta^2$. □

The Bound is Tight. It is natural to wonder whether the obtained bound is tight. The answer is *yes*, as it can be easily seen by evaluating the diverging term $\delta\delta$—please do this exercise. Now, $\delta\delta$ is a diverging term, but it is not hard to obtain a normalising variant. The quadratic bound is indeed reached also by the following family of terms:

$$t_n := (\lambda x_n.(\dots(\lambda x_1.(\lambda x_0.(x_0 x_1 \dots x_n))x_1)x_2 \dots)x_n)I$$

As it is easily seen by running the CbN Micro AM, t_n evaluates in $2n$ β transitions (one for turning each β-redex into an ES, and one for each time that the identity comes in head position) and $\Omega(n^2)$ substitution transitions.

Note that the quadratic overhead is induced by growing chains of substitution steps that only rename variables, induced by sequences of environment entries of the form:

$$[x_1 \leftarrow x_2][x_2 \leftarrow x_3] \dots [x_k \leftarrow x_{k+1}]$$

We discuss at the end how to overcome this issue.

8.2 The High-Level Complexity Analysis of the MAM

Number of Transitions. The bound for the micro-substituting transition $\rightsquigarrow_{sub}$ is an immediate adaptation of the one for the CbN Micro AM.

Lemma 8.3. *Let $r : t_0^\circ \rightsquigarrow_{MAM} Q = (\overline{u}, S, E)$ be a MAM execution. Then:*

1. Micro-substitution linear local bound*: if $r' : Q \rightsquigarrow^*_{sea,sub} Q'$ then $|r'|_{sub} \leq |E| = |r|_\beta$;*
2. Micro-substitution quadratic global bound*: $|r|_{sub} \leq |r|^2_\beta$.*

Proof.

1. Reasoning along the lines of Lemma 5.2 one obtains that $\rightsquigarrow_{sub}$ transitions in r' have to use entries of E from left to right ($\rightsquigarrow_{sea}$ and $\rightsquigarrow_{sub}$ do not modify E), and so $|r'|_{sub} \leq |E|$. Now, $|E|$ is exactly $|r|_\beta$, because the only transition extending E, and of exactly one entry, is $\rightsquigarrow_\beta$.
2. Analogous to the CbN Micro AM case (Lemma 8.2.2). The execution r alternates phases of β-transitions and phases of overhead transitions, i.e. it has the shape:
$$t_0^\circ = Q_1 \rightsquigarrow^*_\beta Q'_1 \rightsquigarrow^*_{sea,sub} Q_2 \rightsquigarrow^*_\beta Q'_2 \rightsquigarrow^*_{sea,sub} \dots Q_k \rightsquigarrow^*_\beta Q'_k \rightsquigarrow^*_{sea,sub} Q$$
Let a_i be the length of the segment $Q_i \rightsquigarrow^*_\beta Q'_i$ and b_i be the number of $\rightsquigarrow_{sub}$ transitions in the segment $Q'_i \rightsquigarrow^*_{sea,sub} Q_{i+1}$, for $i = 1, \dots, k$. Then the same reasoning of Lemma 8.2.2 applies. □

For the searching transition $\rightsquigarrow_{sea}$ the bound relies on the sub-term property. We denote with $|r|_{sea}$ the number of $\rightsquigarrow_{sea}$ transitions in r.

Lemma 8.4. *Let $r : t_0^\circ \rightsquigarrow_{MAM} Q = (\overline{u}, S, E)$ be a MAM execution. Then:*

1. Searching (and β) local bound*: if $r' : Q \rightsquigarrow^*_{\beta,sea} Q'$ then $|r'| \leq |t_0|$;*
2. Searching global bound*: $|r|_{sea} \leq |t_0| \cdot (|r|_{sub} + 1) \leq |t_0| \cdot (|r|^2_\beta + 1)$.*

Proof.

1. The length of r' is bound by the size of the code in the state Q because $\rightsquigarrow_{\beta,sea}$ strictly decreases the size of the code, that in turn is bound by the size $|t_0|$ of the initial term by the sub-term property (Lemma 7.1).
2. The execution r alternates phases of $\rightsquigarrow_\beta$ and $\rightsquigarrow_{sea}$ transitions and phases of $\rightsquigarrow_{sub}$ transitions, i.e. it has the shape:
$$t_0^\circ = Q_1 \rightsquigarrow^*_{\beta,sea} Q'_1 \rightsquigarrow^*_{sub} Q_2 \rightsquigarrow^*_{\beta,sea} Q'_2 \rightsquigarrow^*_{sub} \dots Q_k \rightsquigarrow^*_{\beta,sea} Q'_k \rightsquigarrow^*_{sub} \rightsquigarrow^*_{\beta,sea} Q$$
By Point 1 the length of the segments $Q_i \rightsquigarrow^*_{\beta,sea} Q'_i$ is bound by the size $|t_0|$ of the initial term. The code may grow, instead, with $\rightsquigarrow_{sub}$ transitions. So $|r|_{sea}$ is bound by $|t_0|$ times the number $|r|_{sub}$ of micro-substitution transitions, plus $|t_0|$ once more, because at the beginning there might be $\rightsquigarrow_{\beta,sea}$ transitions before any $\rightsquigarrow_{sub}$ transition—in symbols, $|r|_{sea} \leq |t_0| \cdot (|r|_{sub} + 1)$. Finally, $|t_0| \cdot (|r|_{sub} + 1) \leq |t_0| \cdot (|r|^2_\beta + 1)$ by Lemma 8.3.2. □

Composing the Two Analyses. By composing the analysis of the number of transitions (Lemma 8.4) with the analysis of the cost of single transitions (Lemma **??**) we obtain the complexity of the MAM.

Theorem 8.5 (The MAM is reasonable)**.** *Let $r : t_0^\circ \rightsquigarrow_{MAM} Q$ be a MAM execution. Then:*

1. *$\rightsquigarrow_{sea}$ transitions in r cost all together $\mathcal{O}(|t_0| \cdot (|r|_\beta^2 + 1))$;*
2. *$\rightsquigarrow_\beta$ transitions in r cost all together $\mathcal{O}(|r|_\beta)$;*
3. *$\rightsquigarrow_{sub}$ transitions in r cost all together $\mathcal{O}(|t_0| \cdot (|r|_\beta^2 + 1))$;*

Then r can be implemented on RAM with cost $\mathcal{O}(|t_0| \cdot (|r|_\beta^2 + 1))$, i.e. the MAM is a reasonable implementation of the weak CbN strategy.

Corollary 8.6. *The number of steps of the weak CbN strategy is a reasonable time cost model for the weak λ-calculus.*

Proof. The reasonable simulation of Turing machines by the deterministic λ-calculus—contained in the weak one—is given by Theorem 2.2 in the first lecture. Theorem 6.1 provides a simulation of the weak λ-calculus by the MAM and Theorem 8.5 proves that such a simulation is implementable on Random Access Machines (RAM) in reasonable time. Finally, RAM are reasonably simulated by Turing machines, this is a classic result. □

The Efficient MAM. According to the terminology of Sect. 2, the MAM is reasonable but it is not efficient because micro-substitution takes time quadratic in the length of the strategy. The quadratic factor comes from the fact that in the environment there can be growing chains of renamings, i.e. of substitutions of variables for variables, see [AC14] for more details on this issue. The MAM can actually be optimized easily, obtaining an efficient implementation, by replacing $\rightsquigarrow_\beta$ with the following two *compacting β-transitions*:

Code	*Stack*	*Env*	*Trans*	*Code*	*Stack*	*Env*	
$\lambda x.\overline{t}$	$y :: S$	E	$\rightsquigarrow_{\beta_1}$	$\overline{t}\{x \leftarrow y\}$	S	E	
$\lambda x.\overline{t}$	$\overline{u} :: S$	E	$\rightsquigarrow_{\beta_2}$	$\overline{t}$	S	$[x \leftarrow \overline{u}] :: E$	if $\overline{u}$ is not a variable

Search is Linear and the CbN Micro AM is Reasonable. By Lemma 8.4 the cost of search in the MAM is linear in the number of transitions for implementing micro-substitution. This is an instance of a more general fact: *search* turns out to always be bilinear (in the initial code and in the amount of micro-substitutions). There are two consequences of this general fact. First, it can be turned into a design principle for abstract machines—search *has to be bilinear*, otherwise there is something wrong in the design of the machine. Second, search is somewhat negligible for complexity analyses.

The Searching AM is Unreasonable. It is not hard to see that the Searching AM is unreasonable. Actually, the number of transitions is reasonable. It is indeed easy to prove that the number of searching transitions of the Searching AM is reasonable, by projecting MAM executions on Searching AM executions. The problem is the cost of single β-transitions, that is instead unreasonable. In fact, the Searching AM does not have a sub-term invariant, because it rests on meta-level substitution, and the size of the terms duplicated by the $\rightsquigarrow_\beta$ transition can explode: it is enough to consider the strategy-independent size-exploding family of the first lecture.

The moral is that *micro-substitution is more fundamental than search*. While the cost of search can be expressed in terms of the cost of micro-substitution, the converse is in fact not possible.

9 The MAM in Ocaml

We want now to show an possible implementation of the MAM in Ocaml, due to Accattoli and Barras [AB17]. The mainstream approach for implementing abstract machines is to use so called *de Bruijn indices*. We here present an alternative approach, hinted at during the lecture, where variables and environments are represented via a store. The aim is twofold. On the one hand, we want to show that our informal hypotheses about the concrete implementation of the Ocaml are justified, presenting an implementation respecting the cost of single transitions claimed in the previous section. On the other hand, we aim at making more popular an interesting and less known way of implementing the λ-calculus.

Terms and States. The first step is the following definition of the type of terms.

```
type term =
    Var of var          (* Variable occurrences*)
 |  App of term * term        (* Applications *)
 |  Lam of var * term         (* Abstractions *)
and var = { name:string; mutable subs:subs }
and subs = NotSub | Subs of term | Copy of var
```

A variable has a string `name` field, which is used just for printing, meaning that equality on variables is *pointer* equality, not name equality. The mutable field `subs` has three possible values—let's focus on the first two and ignore `Copy` for the moment. To ensure the soundness of the term representation, the follow- ing invariant needs to be enforced: the variable of an abstraction must be in the `NotSub` status. The `Subs` status of a variable x encodes the fact that there is an explicit substitution $[x \leftarrow t]$ on x.

A state is given by a term and a stack:

```
type state = term * term list
```

The idea is that the MAM environment E is *not* explicitly part of a state, because it is simply given by all variables with an explicit substitution on them, that is, in `Subs` status.

Transitions. The transitions of the MAM are implemented as follows:

```
let rec mam (st:state) : state =
  match st with
  | App(t,u),          stk -> mam (t,u::stk)
  | Lam(x, t), (u::stk) ->
     x.subs <- Subs u;
     mam (t,stk)
  | Var{subs=Subs t}, stk -> mam (copy t,stk)
  | (Lam _, [] | Var _, _) -> st
```

So the transition is selected by pattern matching on the term part of the state `st`. The first transition corresponds to $\rightsquigarrow_{sea}$. The second one corrsponds to $\rightsquigarrow_{\beta}$. Note that it puts the argument on the stack inside the ES associated to `x`. The third transition corresponds to $\rightsquigarrow_{sub}$. Note that it needs not traversing the environment sequentially (because there is no sequential encoding of the environment) and that it relies on a copy function—corresponding to the α-renaming $\overline{t}^{\alpha}$ operation in $\rightsquigarrow_{sub}$—to be discussed next. The last case is for the two possible final states of an abstraction coming with an empty stack and of a variable without an associated ES, and these cases are handled by simply returning the final state.

Note that the function implementing the transitions is tail recursive, which guarantees that its recursion is not space consuming.

Naive Copying. The copy function used in the implementation of the substitution transition $\rightsquigarrow_{sub}$ requires some care in order to be implemented in time linear in the size of the copied term. The naive copy algorithm, that we now see, indeed has quadratic complexity. We shall then refine it into a linear one.

The following naive copy algorithm `rename` traverses the term `t` to copy allocating a new variable `y` at every binder `Lam(x,u)` and propagating the list `renMap` of generated renamings `(x,y)` as to replace every variable occurrence accordingly.

```
let rec rename renMap t =
  match t with
  | App(u,v) -> App(rename renMap u, rename renMap v)
  | Lam(x,u) ->
     let y = mkvar x.name in
     Lam(y,rename ((x,y)::renMap) u)
  | Var x ->
     (try Var (List.assq x renMap)
      with Not_found -> t)
let copy = rename []
```

The exception `Not_found` is for occurrences of free variables.

The complexity of `rename` is easily seen to be quadratic, as we now explain. The number of binders in `t` is $\mathcal{O}(|\mathtt{t}|)$, implying that the size of `renMap` is $\mathcal{O}(|\mathtt{t}|)$. The number

of variable occurrences in `t` is also $\mathcal{O}(|\texttt{t}|)$, each one requiring to look up `renMap`, giving total cost $\mathcal{O}(|\texttt{t}|^2)$.

Linear Time Copying. To improve the complexity of copying we need to get rid of the list of renamings `renMap`. This can be done with a trick, exploiting ES in a way that goes beyond their specification in the MAM and making use of the `Copy` status of variables that we ignored at the beginning. The idea is that when the new efficient copy function `effcopy` below finds a binder `Lam(x,u)`, it allocates a new variable `y` add puts it as the content of the `Copy` status of `x`. Then `u` is copied recursively, replacing the occurrences of `x` with occurrences of `y`, retrieved via the `Copy` field. When the efficient copy of `u` returns, the status of `x` is set back to `Not_Sub`. Essentially, `Copy` is a temporary marker for a variable `x` which is active only during the copy of the body `u` of the abstraction `Lam(x,u)` of `x`.

```
let rec effcopy t =
  match t with
  | App(u,v) -> App(effcopy u, effcopy v)
  | Lam(x,u) ->
     let y = mkvar x.name in
     x.subs <- Copy y;
     let uWithXRenamedY = effcopy u in
     x.subs <- NotSub;
     Lam(y,uWithXRenamedY)
  | Var{subs=Copy y} -> Var y
  | Var _ -> t
```

The copy of variable occurrences in `effcopy t` requires constant time and so the complexity is linear in $|\texttt{t}|$. That is, the given code validates the complexity assumptions for the MAM, showing that it can realistically be implemented with linear-cost steps.

References

[AB17] Beniamino Accattoli and Bruno Barras. Environments and the complexity of abstract machines. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP 2017)*, pages 4–16, 2017.

[AC14] Beniamino Accattoli and Claudio Sacerdoti Coen. On the value of variables. In *WoLLIC 2014*, pages 36–50, 2014.