ASMs and Operational Algorithmic Completeness of Lambda Calculus

Marie Ferbus-Zanda [*]
ferbus@liafa.jussieu.fr

Serge Grigorieff [*]
http://www.liafa.jussieu.fr/∼seg
seg@liafa.jussieu.fr

September 13, 2018

Contents

[*]LIAFA, CNRS & Université Paris Diderot - Paris 7, Case 7014 75205 Paris Cedex 13

Abstract

We show that lambda calculus is a computation model which can step by step simulate any sequential deterministic algorithm for any computable function over integers or words or any datatype. More formally, given an algorithm above a family of computable functions (taken as primitive tools, i.e., kind of oracle functions for the algorithm), for every constant K big enough, each computation step of the algorithm can be simulated by exactly K successive reductions in a natural extension of lambda calculus with constants for functions in the above considered family.
The proof is based on a fixed point technique in lambda calculus and on Gurevich sequential Thesis which allows to identify sequential deterministic algorithms with Abstract State Machines.
This extends to algorithms for partial computable functions in such a way that finite computations ending with exceptions are associated to finite reductions leading to terms with a particular very simple feature.

keywords. ASM, Lambda calculus, Theory of algorithms, Operational semantics

1 Introduction

1.1 Operational versus Denotational Completeness

Since the pioneering work of Church and Kleene, going back to 1935, many computation models have been shown to compute the same class of functions, namely, using Turing Thesis, the class of all computable functions. Such classes are said to be *Turing complete* or *denotationally algorithmically complete.*

This is a result about crude input/output behaviour. What about the ways to go from the input to the output, i.e., the executions of algorithms in each of these computation models? Do they constitute the same class? Is there a Thesis for algorithms analog to Turing Thesis for computable functions?

As can be expected, denotational completeness does not imply operational completeness. Clearly, the operational power of machines using massive parallelism cannot be matched by sequential machines. For instance, on networks of cellular automata, integer multiplication can be done in real time (cf. Atrubin, 1962 [1], see also Knuth, [21] p.394-399), whereas on Turing machines, an $\Omega(n/\log n)$ time lower bound is known. Keeping within sequential computation models, multitape Turing machines have greater operational power than one-tape Turing machines. Again, this is shown using a complexity argument: palindromes recognition can be done in linear time on two-tapes Turing machines, whereas it requires computation time $O(n^2)$ on one-tape Turing machines (Hennie, 1965 [18], see also [5, 24]).

Though resource complexity theory may disprove operational algorithmic completeness, there was no formalization of a notion of operational completeness since the notion of algorithm itself had no formal mathematical modelization. Tackled by Kolmogorov in the 50's [20], the question for *sequential algorithms* has been answered by Gurevich in the 80's [11, 12, 13] (see [6] for a comprehensive survey of the question), with their formalization as "*evolving algebras*" (now called *"abstract state machines" or ASMs*) which has lead to *Gurevich's sequential Thesis.*

Essentially, an ASM can be viewed as a first order multi-sorted structure and a program which modifies some of its predicates and functions (called dynamic items). Such dynamic items capture the moving environment of a procedural program. The run of an ASM is the sequence of structures – also called states – obtained by iterated application of the program. The program itself includes two usual ingredients of procedural languages, namely affectation and the conditional "if... then... else...", plus a notion of parallel block of instructions. This last notion is a key idea which is somehow a programming counterpart to the mathematical notion of system of equations.

Gurevich's sequential Thesis [12, 16, 17] asserts that ASMs capture the notion of sequential algorithm. Admitting this Thesis, the question of operational completeness for a sequential procedural computation model is now the comparison of its operational power with that of ASMs.

1.2 Lambda Calculus and Operational Completeness

In this paper we consider lambda calculus, a subject created by Church and Kleene in the 30's, which enjoys a very rich mathematical theory. It may seem a priori strange to look for operational completeness with such a computation model so close to an assembly language (cf. Krivine's papers since 1994, e.g., [22]). It turns out that, looking at reductions by groups (with an appropriate but constant length), and allowing one step reduction of primitive operations, lambda calculus simulates ASMs in a very tight way. Formally, our translation of ASMs in lambda calculus is as follows. Given an ASM, we prove that, for every integer K big enough (the least such K depending on the ASM), there exists a lambda term θ with the following property. Let $a_1^t, \dots, a_p^t$ be the values (coded as lambda terms) of all dynamic items of the ASM at step t, if the run does not stop at step t then

$$\theta a_1^t \dots a_p^t \quad \overbrace{\rightarrow \quad \cdots \quad \rightarrow}^{K \text{ reductions}} \quad \theta a_1^{t+1} \dots a_p^{t+1} \ .$$

If the run stops at step t then the left term reduces to a term in normal form which gives the list of outputs if they are defined. Thus, representing the state of the ASM at time t by the term $\theta a_1^t \dots a_p^t$, a group of K successive reductions gives the state at time $t+1$. In other words, K reductions faithfully simulate one step of the ASM run. Moreover, this group of reductions is that obtained by the *leftmost redex reduction strategy*, hence it is a deterministic process. Thus, *lambda calculus is operationally complete for deterministic sequential computation.*

Let us just mention that adding to lambda calculus one step reduction of primitive operations is not an unfair trick. Every algorithm has to be "above" some basic operations which are kind of oracles: the algorithm decomposes the computation in elementary steps which are considered as atomic steps though they obviously themselves require some work. In fact, such basic operations can be quite complex: when dealing with integer matrix product (as in Strassen's algorithm in time $O(n^{\log 7})$), one considers integer addition and multiplication as basic... Building algorithms on such basic operations is indeed what ASMs do with the so-called static items, cf. §2.3, Point 2.

The proof of our results uses Curry's fixed point technique in lambda calculus plus some padding arguments.

1.3 Road Map

This paper deals with two subjects which have so far not been much related: ASMs and lambda calculus. To make the paper readable to both ASM and lambda calculus communities, the next two sections recall all needed prerequisites in these two domains (*so that most readers may skip one of these two sections*).

What is needed about ASMs is essentially their definition, but it cannot be given without a lot of preliminary notions and intuitions. Our presentation of ASMs in §2 differs in inessential ways from Gurevich's one (cf. [13, 15, 17, 10]). Crucial in the subject (and for this paper) is Gurevich's sequential Thesis that we state in §2.2. We rely on the literature for the many arguments supporting this Thesis.

§3 recalls the basics of lambda calculus, including the representation of lists and integers and Curry fixed point combinator.

The first main theorem in §5.3 deals with the simulation in lambda calculus of sequential algorithms associated to ASMs in which all dynamic symbols are constant ones (we call them type 0 ASMs). The second main theorem in §5.4 deals with the general case.

Euclid's algorithm in Pascal

```
while $b > 0$ do begin
                    $z := a$;
                    $a := b$;
                    $b := rem\ (z, b)$;
                    end;
$gcd := a$.
```

Euclid's algorithm in ASM

```
if $0 < b$ then | $a := b$
                | $b := rem\ (a, b)$
```

(In both programs, a, b are inputs and a is the output)

Figure 1: Pascal and ASM programs for Euclid's algorithm

2 ASMs

2.1 The Why and How of ASMs on a Simple Example

Euclid's Algorithm Consider Euclid's algorithm to compute the greatest common divisor (gcd) of two natural numbers. It turns out that such a simple algorithm already allows to pinpoint an operational incompleteness in usual programming languages. Denoting by $\texttt{rem}(u, v)$ the remainder of u modulo v, this algorithm can be described as follows[1]

```
Given data: two natural numbers $a, b$
While $b \neq 0$ replace the pair $(a, b)$ by $(b, \texttt{rem}(a, b))$
When $b = 0$ halt: $a$ is the wanted gcd
```

Observe that the the pair replacement in the above while loop involves some elementary parallelism which is the algorithmic counterpart to co-arity, i.e., the consideration of functions with range in multidimensional spaces such as the $\mathbb{N}^2 \to \mathbb{N}^2$ function $(x, y) \mapsto (y, \texttt{rem}(x, y))$.

Euclid's Algorithm in Pascal In usual programming languages, the above simultaneous replacement is impossible: affectations are not done in parallel but sequentially. For instance, *no Pascal program implements it as it is*, one can only get a distorted version with an extra algorithmic contents involving a new variable z, cf. Figure 1.

[1]Sometimes, one starts with a conditional swap: if $a < b$ then a, b are exchanged. But this is done in the first round of the while loop.

An ASM for Euclid's Algorithm Euclid's algorithm has a faithful formalization using an ASM. The vertical bar on the left in the ASM program (cf. Figure 1) tells that the two updates are done simultaneously and independently. Initialization gives symbols a, b the integer values of which we want to compute the gcd. The semantical part of the ASM involves the set $\mathbb{N}$ of integers to interpret all symbols. Symbols $0, <, =, rem$ have fixed interpretations in integers which are the expected ones. Symbols a, b have varying interpretations in the integers. The sequence of values taken by a, b constitutes the run of the ASM.
When the instruction gets void (i.e., when b is null) the run stops and the value of the symbol a is considered to be the output.

2.2 Gurevich Sequential Thesis

Yuri Gurevich has gathered as three Sequential Postulates (cf. [17, 10]) some key features of deterministic sequential algorithms for partial computable functions (or type 1 functionals).

I *(Sequential time).* An algorithm is a deterministic state-transition system. Its transitions are partial functions.
Non deterministic transitions and even nonprocedural input/output specifications are thereby excluded from consideration.

II *(Abstract states).* States are multitructures[2], sharing the same fixed, finite vocabulary. States and initial states are closed under isomorphism. Transitions preserve the domain, and transitions and isomorphisms commute.

III *(Bounded exploration).* Transitions are determined by a fixed finite "glossary" of "critical" terms. That is, there exists some finite set of (variable-free) terms over the vocabulary of the states such that states that agree on the values of these glossary terms also agree on all next-step state changes.

Gurevich, 2000 [17], stated an operational counterpart to Church's Thesis : **Thesis.**[Gurevich's sequential Thesis] *E*very sequential algorithm satisfies the Sequential Postulates I-III.

[2]In ASM theory, an ASM is, in fact, a multialgebra (cf. point 1 of Remark §2.1).

2.3 The ASM Modelization Approach

Gurevich's postulates lead to the following modelization approach (we depart in non essential ways from [10], see Remark 2.1).

1. *The base sets.* Find out the underlying families of objects involved in the given algorithm, i.e., objects which can be values for inputs, outputs or environmental parameters used during the execution of the algorithm. These families constitute the base sets of the ASM. In Euclid's algorithm, a natural base set is the set $\mathbb{N}$ of natural integers.

2. *Static items.* Find out which particular fixed objects in the base sets are considered and which functions and predicates over/between the base sets are viewed as atomic in the algorithm, i.e., are not given any modus operandi. Such objects, functions and predicates are called the primitive or static items of the ASM. They do not change value through transitions. In Euclid's algorithm, static items are the integer 0, the *rem* function and the $<$ predicate.

3. *Dynamic items.* Find out the diverse objects, functions and predicates over the base sets of the ASM which vary through transitions. Such objects, functions and predicates are called the dynamic items of the ASM. In Euclid's algorithm, these are a, b.

4. *States: from a multi-sorted partial structure to a multi-sorted partial algebra.* Collecting all the above objects, functions and predicates leads to a first-order multi-sorted structure of some logical typed language: any function goes from some product of sorts into some sort, any predicate is a relation over some sorts. However, there is a difference with the usual logical notion of multi-sorted structure: predicates and functions may be partial. A feature which is quite natural for any theory of computability, a fortiori for any theory of algorithms.
   To such a multi-sorted structure one can associate a multi-sorted algebra as follows. First, if not already there, add a sort for Booleans. Then replace predicates by their characteristic functions In this way, we get a multi-sorted structure with partial functions only, i.e. a multialgebra.

5. *Programs.* Finally, the execution of the algorithm can be viewed as a sequence of states. Going from one state to the next one amounts

to applying to the state a particular program – called the ASM program – which modifies the interpretations of the sole dynamic symbols (but the universe itself and the interpretations of the static items remain unchanged). Thus, the execution of the algorithm appears as an iterated application of the ASM program. It is called the run of the ASM.
Using the three above postulates, Gurevich [16, 17] proves that quite elementary instructions – namely blocks of parallel conditional updates – suffice to get ASM programs able to simulate step by step any deterministic procedural algorithm.

6. *Inputs, initialization map and initial state.* Inputs correspond to the values of some distinguished static symbols in the initial state, i.e., we consider that all inputs are given when the algorithm starts (though questionable in general, this assumption is reasonable when dealing with algorithms to compute a function). All input symbols have arity zero for algorithms computing functions. Input symbols with non zero arity are used when dealing with algorithms for type 1 functionals.
The initialization map associates to each dynamic symbol a term built up with static symbols. In an initial state, the value of a dynamic symbol is required to be that of the associated term given by the initialization map.

7. *Final states and outputs.* There may be several outputs, for instance if the algorithm computes a function $\mathbb{N}^k \to \mathbb{N}^\ell$ with $\ell \geq 2$.
A state is final when, applying the ASM program to that state,

   (a) either the `Halt` instruction is executed *(Explicit halting)*,

   (b) or no update is made (i.e. all conditions in conditional blocks of updates get value *False*) *(Implicit halting)* .

   In that case, the run stops and the outputs correspond to the values of some distinguished dynamic symbols. For algorithms computing functions, all output symbols are constants (i.e. function symbols with arity zero).

8. *Exceptions.* There may be a finite run of the ASM ending in a non final state. This corresponds to exceptions in programming (for instance a division by 0) and there is no output in such cases. This happens when

(a) either the `Fail` instruction is executed *(Explicit failing)*,

(b) or there is a clash between two updates which are to be done simultaneously *(Implicit failing)*.

Remark 2.1. Let us describe how our presentation of ASMs (slightly) departs from [10].
1. We stick to what Gurevich says in §.2.1 of [14] (Lipari Guide, 1993): *"Actually, we are interested in multi-sorted structures with partial operations"*. Thus, we do not regroup sorts into a single universe and do not extend functions with the *undef* element.
2. We add the notion of initialization map which brings a syntactical counterpart to the semantical notion of initial state. It also rules out any question about the status of initial values of dynamic items which would not be inputs.
3. We add explicit acceptance and rejection as specific instructions in ASM programs. Of course, they can be simulated using the other ASM instructions (so, they are syntactic sugar) but it may be convenient to be able to explicitly tell there is a failure when something like a division by zero is to be done. This is what is done in many programming languages with the so-called exceptions. Observe that `Fail` has some common flavor with `undef`. However, `Fail` is relative to executions of programs whereas `undef` is relative to the universe on which the program is executed.
4. As mentioned in §2.1, considering several outputs goes along with the idea of parallel updates.

2.4 Vocabulary and States of an ASM

ASM vocabularies and ASM states correspond to algebraic signatures and algebras. The sole difference is that an ASM vocabulary comes with an extra classification of its symbols as static, dynamic, input and output carrying the intuitions described in points 2, 3, 6, 7 of §2.3.

Definition 2.2. *1. An ASM vocabulary is a finite family of sorts $s_1, \ldots, s_m$ and a finite family $\mathcal{L}$ of function symbols with specified types of the form s_i or $s_{i_1} \times \cdots \times s_{i_k} \to s_i$ (function symbols with type s_i are also called constants of type s_i). Four subfamilies of symbols are distinguished:*

$$\begin{array}{lcl} \mathcal{L}^{sta} \text{ (static symbols)} & , & \mathcal{I} \text{ (input symbols)} \\ \mathcal{L}^{dyn} \text{ (dynamic symbols)} & , & \mathcal{O} \text{ (output symbols)} \end{array}$$

such that $\mathcal{L}^{sta}, \mathcal{L}^{dyn}$ is a partition of $\mathcal{L}$ and $\mathcal{I} \subseteq \mathcal{L}^{sta}$ and $\mathcal{O} \subseteq \mathcal{L}^{dyn}$. We also require that there is a sort to represent Booleans and that $\mathcal{L}^{sta}$ contains symbols to represent the Boolean items (namely symbols `True`*,* `False`*, $\neg$, $\wedge$, $\vee$) and, for each sort s, a symbol $=_s$ to represent equality on sort s.*

*2. Let $\mathcal{L}$ be an ASM vocabulary with n sorts. An $\mathcal{L}$-*state *is any n-sort multialgebra $\mathcal{S}$ for the vocabulary $\mathcal{L}$. The multi-domain of $\mathcal{S}$ is denoted by $(\mathcal{U}_1, \ldots, \mathcal{U}_m)$. We require that*

- *i. one of the $\mathcal{U}_i$'s is* `Bool` *with the expected interpretations of symbols* `True`*,* `False`*,s $\neg$, $\wedge$, $\vee$,*
- *ii. the interpretation of the symbol $=_i$ is usual equality in the interpretation $\mathcal{U}_i$ of sort s_i.*

In the usual way, using variables typed by the n sorts of $\mathcal{L}$, one constructs typed $\mathcal{L}$-terms and their types. The type of a term t is of the form s_i or $s_{i_1} \times \cdots \times s_{i_k} \to s_i$ where $s_{i_1}, \ldots, s_{i_k}$ are the types of the different variables occurring in t. Ground terms are those which contain no variable. The semantics of typed terms is the usual one.

Definition 2.3. *Let $\mathcal{L}$ be an ASM vocabulary and $\mathcal{S}$ an ASM $\mathcal{L}$-state. Let t be a typed term with type $s_{i_1} \times \cdots \times s_{i_1} \to s_i$. We denote by $t_{\mathcal{S}}$ its interpretation in $\mathcal{S}$, which is a function $\mathcal{U}_{i_1} \times \cdots \times \mathcal{U}_{i_\ell} \to \mathcal{U}_i$. In case $\ell = 0$, i.e., no variable occurs, then $t_{\mathcal{S}}$ is an element of $\mathcal{U}_i$.*

It will be convenient to lift the interpretation of a term with ℓ variables to be a function with any arity k greater than ℓ.

Definition 2.4. *Let $\mathcal{L}$ be an ASM vocabulary and $\mathcal{S}$ an ASM $\mathcal{L}$-state with universe $\mathcal{U}$. Suppose $\sigma : \{1, \ldots, \ell\} \to \{1, \ldots, p\}$ is any map and $\tau : \{1, \ldots, p\} \to \{1, \ldots, m\}$ is a distribution of (indexes of) sorts. Suppose t is a typed term of type $s_{\tau(\sigma(1))} \times \cdots \times s_{\tau(\sigma(\ell))} \to s_i$. We let $t^{\tau,\sigma}_{\mathcal{S}}$ be the function $\mathcal{U}_{s_{\tau(1)}} \times \cdots \times \mathcal{U}_{s_{\tau(p)}} \to \mathcal{U}_i$ such that, for all $(a_1, \cdots, a_p) \in \mathcal{U}_{s_{\tau(1)}} \times \cdots \times \mathcal{U}_{s_{\tau(p)}}$,*

$$t^{\tau,\sigma}_{\mathcal{S}}(a_1, \cdots, a_k) \quad = \quad t_{\mathcal{S}}(a_{\sigma(1)}, \cdots, a_{\sigma(\ell)}) \ .$$

2.5 Initialization Maps

$\mathcal{L}$-terms with no variable are used to name particular elements in the universe $\mathcal{U}$ of an ASM whereas $\mathcal{L}$-terms with variables are used to name particular functions over $\mathcal{U}$.

Using the lifting process described in Definition 2.4, one can use terms containing less than k variables to name functions with arity k.

Definition 2.5. *1, Let $\mathcal{L}$ be an ASM vocabulary. An $\mathcal{L}$-*initialization map *ξ has domain family $\mathcal{L}^{(dyn)}$ of dynamic symbols and satisfies the following condition:*

> *if α is a dynamic function symbol with type $s_{\tau(1)} \times \cdots \times s_{\tau(\ell)} \to s_i$ then $\xi(\alpha)$ is a pair (σ, t) such that $\sigma : \{1, \ldots, \ell\} \to \{1, \ldots, p\}$ and t is a typed $\mathcal{L}$-term with type $s_{\tau(\sigma(1))} \times \cdots \times s_{\tau(\sigma(\ell))} \to s_i$ which is built with the sole static symbols (with $\tau : \{1, \ldots, p\} \to \{1, \ldots, m\}$).*

2. Let ξ be an $\mathcal{L}$-initialization map. An $\mathcal{L}$-state $\mathcal{S}$ is ξ-initial if, for any dynamic function symbol α, if $\xi(\alpha) = (\sigma, t)$ then the interpretation of α in $\mathcal{S}$ is $t_{\mathcal{S}}^{\tau,\sigma}$.

3. An $\mathcal{L}$-state is initial if it is ξ-initial for some ξ.

Remark 2.6. Of course, the values of static symbols are basic ones, they are not to be defined from anything else: either they are inputs or they are the elementary pieces upon which the ASM algorithm is built.

2.6 ASM Programs

Definition 2.7. *1. The vocabulary of ASM programs is the family of symbols*

$$\{\texttt{Skip}\ ,\ \texttt{Halt}\ ,\ \texttt{Fail}\ ,\ :=\ ,\ \Big|\ ,\ \texttt{if}\ \ldots\ \texttt{then}\ \ldots\ \texttt{else}\ \ldots\}$$

2. ($\mathcal{L}$-updates). *Given an ASM vocabulary $\mathcal{L}$, a sequence of $k+1$ ground typed $\mathcal{L}$-terms $t_1, \ldots, t_k, u$ (i.e. typed terms with no variable), a dynamic function symbol α, if $\alpha(t_1, \ldots, t_k)$ is a typed $\mathcal{L}$-term with the same type as u then the syntactic object $\alpha(t_1, \ldots, t_k) := u$ is called an $\mathcal{L}$-update.*

3. ($\mathcal{L}$-programs). *Given an ASM vocabulary $\mathcal{L}$, the $\mathcal{L}$ programs are obtained via the following clauses.*

i. (Atoms). `Skip`, `Halt`, `Fail` *and all $\mathcal{L}$-updates are $\mathcal{L}$-programs.*

ii. (Conditional constructor). *Given a ground typed term C with Boolean type and two $\mathcal{L}$-programs P, Q, the syntactic object*

$$\texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q$$

is an $\mathcal{L}$-program.

iii. (Parallel block constructor). *Given $n \geq 1$ and $\mathcal{L}$-programs $P_1, \ldots, P_n$, the syntactic object (with a vertical bar on the left)*

$$\left| \begin{array}{l} P_1 \\ \vdots \\ P_n \end{array} \right.$$

is an $\mathcal{L}$-program.

The intuition of programs is as follows.

- `Skip` is the program which does nothing. `Halt` halts the execution in a successful mode and the outputs are the current values of the output symbols. `Fail` also halts the execution but tells that there is a failure, so that there is no meaningful output.
- Updates modify the interpretations of dynamic symbols, they are the basic instructions. The left member has to be of the form $\alpha(\cdots)$ with α a dynamic symbol because the interpretations of static symbols do not vary.
- The conditional constructor has the usual meaning whereas the parallel constructor is a new control structure to get *simultaneous and independent executions* of programs $P_1, \ldots, P_n$.

2.7 Action of an $\mathcal{L}$-Program on an $\mathcal{L}$-State

2.7.1 Active Updates and Clashes

In a program the sole instructions which have some impact are updates. They are able to modify the interpretations of dynamic symbols on the sole tuples of values which can be named by tuples of ground terms. Due to conditionals, not every update occurring in a program will really be active. it does depend on the state to which the program is applied. Which symbols on which tuples are really active and what is their action? This is the object of the next definition.

Definition 2.8 (Active updates). *Let $\mathcal{L}$ be an ASM vocabulary, P an $\mathcal{L}$-program and $\mathcal{S}$ an $\mathcal{L}$-state. Let* `Update`(P) *be the family of all updates occur-*

ring in P. The subfamily $\texttt{Active}\ (\mathcal{S}, P) \subseteq \texttt{Update}(P)$ *of so-called* $(\mathcal{S}, P)$-active updates *is defined via the following induction on P :*

$$\texttt{Active}\ (\mathcal{S}, \texttt{Skip}) = \emptyset$$

$$\texttt{Active}\ (\mathcal{S}, \alpha(t_1, \ldots, t_k) := u) = \{\alpha(t_1, \ldots, t_k) := u\}$$

$$\texttt{Active}\ (\mathcal{S},\ \texttt{if}\ C\ \texttt{then}\ Q\ \texttt{else}\ R) = \begin{cases} \texttt{Active}\ (\mathcal{S}, Q) & \text{if } C_{\mathcal{S}} = \texttt{True} \\ \texttt{Active}\ (\mathcal{S}, R) & \text{if } C_{\mathcal{S}} = \texttt{False} \\ \emptyset & \text{if } C_{\mathcal{S}} \notin \texttt{Bool} \end{cases}$$

$$\texttt{Active}\ (\mathcal{S}, \left| \begin{array}{l} P_1 \\ \vdots \\ P_n \end{array} \right.) = \texttt{Active}\ (\mathcal{S}, P_1) \cup \ldots \cup \texttt{Active}\ (\mathcal{S}, P_n)$$

The action of a program P on a state $\mathcal{S}$ is to be seen as the conjunction of updates in $\texttt{Active}\ (\mathcal{S}, P)$ provided these updates are compatible. Else, P clashes on $\mathcal{S}$.

Definition 2.9. *An $\mathcal{L}$-program P clashes on an $\mathcal{L}$-state $\mathcal{S}$ if there exists two active updates $\alpha(s_1, \ldots, s_k) := u$ and $\alpha(t_1, \ldots, t_k) := v$ in* $\texttt{Active}\ (\mathcal{S}, P)$ *relative to the same dynamic symbol α such that $s_{1\mathcal{S}} = t_{1\mathcal{S}}$, $\ldots$, $s_{k\mathcal{S}} = t_{k\mathcal{S}}$ but $u_{\mathcal{S}}$ and $v_{\mathcal{S}}$ are not equal (as elements of the universe).*

Remark 2.10. A priori, another case could also be considered as a clash. We illustrate it for a parallel block of two programs P, Q and the update of a dynamic constant symbol c. Suppose $c_{\mathcal{S}} \neq u_{\mathcal{S}}$ and $c := u$ is an active update in $\texttt{Active}\ (\mathcal{S}, P)$. Then P wants to modify the value of $c_{\mathcal{S}}$. Suppose also that there is no active update with left member c in $\texttt{Active}\ (\mathcal{S}, Q)$. Then Q does not want to touch the value of $c_{\mathcal{S}}$. Thus, P and Q have incompatible actions: P modifies the interpretation of c whereas Q does nothing about c. One could consider this as a clash for the parallel program $\left| \begin{array}{l} P \\ Q \end{array} \right.$. Nevertheless, *this case is not considered to be a clash.* A moment reflection shows that this is a reasonable choice. Otherwise, a parallel block would always clash except in case all programs $P_1, \ldots, P_n$ do exactly the same actions... Which would make parallel blocks useless.

2.7.2 Halt and Fail

Definition 2.11. *Let $\mathcal{L}$ be an ASM vocabulary, $\mathcal{S}$ be an $\mathcal{L}$-state and P an $\mathcal{L}$-program. By induction, we define the two notions: P halts (resp. fails) on $\mathcal{S}$.*

- *If P is* `Skip` *or an update then P neither halts nor fails on $\mathcal{S}$.*
- *If P is* `Halt` *(resp.* `Fail`*) then P halts and does not fail (resp. fails and does not halt) on $\mathcal{S}$.*
- `if` C `then` Q `else` R *halts on $\mathcal{S}$ if and only if*

$$\begin{cases} \textit{either } C_{\mathcal{S}} = \texttt{True} \textit{ and } Q \textit{ halts on } \mathcal{S} \\ \textit{or } C_{\mathcal{S}} = \texttt{False} \textit{ and } R \textit{ halts on } \mathcal{S} \end{cases}$$

- `if` C `then` Q `else` R *fails on $\mathcal{S}$ if and only if*

$$\begin{cases} \textit{either } C_{\mathcal{S}} = \texttt{True} \textit{ and } Q \textit{ fails on } \mathcal{S} \\ \textit{or } C_{\mathcal{S}} = \texttt{False} \textit{ and } R \textit{ fails on } \mathcal{S}\ . \end{cases}$$

- *The parallel block of programs $P_1, \ldots, P_n$ halts on $\mathcal{S}$ if and only if some P_i halts on $\mathcal{S}$ and no P_j fails on $\mathcal{U}$.*
- *The parallel block of programs $P_1, \ldots, P_n$. fails on $\mathcal{S}$ if and only if some P_i fails on $\mathcal{S}$.*

2.7.3 Successor State

Definition 2.12. Let $\mathcal{L}$ be an ASM vocabulary and $\mathcal{S}$ be an $\mathcal{L}$-state.
The *successor state* $\mathcal{T} = \texttt{Succ}(\mathcal{S}, P)$ of state $\mathcal{S}$ relative to an $\mathcal{L}$-program P is defined if only if P does not clash nor fail nor halt on $\mathcal{S}$.
In that case, the successor is inductively defined via the following clauses.

1. $\mathcal{T} = \texttt{Succ}(\mathcal{S}, P)$ and $\mathcal{S}$ have the same base sets $\mathcal{U}_1, \ldots, \mathcal{U}_n$.
2. $\alpha_{\mathcal{T}} = \alpha_{\mathcal{S}}$ for any static symbol α.

3a. $\texttt{Succ}(\mathcal{S}, \texttt{Skip}) = \mathcal{S}$ (recall that `Skip` does nothing. . . .)

3b. Suppose P is an update program $\alpha(t_1, \ldots, t_k) := u$ where α is a dynamic symbol with type $s_{i_1} \times \cdots \times s_{i_k} \to s_i$ and $\vec{a} = (t_{1\mathcal{S}}, \ldots, t_{k\mathcal{S}})$. Then all dynamic symbols different from α have the same interpretation in $\mathcal{S}$ and $\mathcal{T}$ and, for every $\vec{b} \in \mathcal{U}_{i_1} \times \cdots \times \mathcal{U}_{i_k}$, we have $\alpha_{\mathcal{T}}(\vec{b}) = \begin{cases} \alpha_{\mathcal{S}}(\vec{b}) & \text{if } \vec{b} \neq \vec{a} \\ u_{\mathcal{S}} & \text{if } \vec{b} = \vec{a} \end{cases}$.

3c. Suppose P is the conditional program `if` C `then` Q `else` R. Then

$$\begin{cases} \mathtt{Succ}(\mathcal{S},P)=\mathtt{Succ}(\mathcal{S},Q) & \text{if } C_{\mathcal{S}}=\mathtt{True} \\ \mathtt{Succ}(\mathcal{S},P)=\mathtt{Succ}(\mathcal{S},R) & \text{if } C_{\mathcal{S}}=\mathtt{False} \end{cases}$$

(since P does not fail on $\mathcal{S}$, we know that $C_{\mathcal{S}}$ is a Boolean).

3d Suppose P is the parallel block program $\left| \begin{array}{l} P_1 \\ \vdots \\ P_n \end{array} \right.$ and P does not clash on $\mathcal{S}$. Then $\mathcal{T}=\mathtt{Succ}(\mathcal{S},P)$ is such that, for every dynamic symbol α with type $s_{i_1}\times\cdots\times s_{i_k}\to s_i$ and every tuple $\vec{a}=(a_1,\ldots,a_k)$ in $\mathcal{U}_{i_1}\times\cdots\times\mathcal{U}_{i_k}$,

- if there exists an update $\alpha(t_1,\ldots,t_k):=u$ in `Active` $(\mathcal{S},P)$ such that $\vec{a}=(t_{1\mathcal{S}},\ldots,t_{k\mathcal{S}})$ then $\alpha(\vec{a})_{\mathcal{T}}$ is the common value of all $v_{\mathcal{S}}$ for which there exists some update $\alpha(s_1,\ldots,s_k):=v$ in `Active` $(\mathcal{S},P)$ such that $\vec{a}=(s_{1\mathcal{S}},\ldots,s_{k\mathcal{S}})$.
- Else $\alpha(\vec{a})_{\mathcal{T}}=\alpha(\vec{a})_{\mathcal{S}}$.

Remark 2.13. In particular, $\alpha_{\mathcal{T}}(\vec{a})$ and $\alpha_{\mathcal{S}}(\vec{a})$ have the same value in case $\vec{a}=(a_1,\ldots,a_k)$ is not the value in $\mathcal{S}$ of any k-tuple of ground terms $(t_1,\ldots,t_k)$ such that `Active` $(\mathcal{S},P)$ contains an update of the form $\alpha(t_1,\ldots,t_k):=u$ for some ground term u.

2.8 Definition of ASMs and ASM Runs

At last, we can give the definition of ASMs and ASM runs.

Definition 2.14. *1. An ASM is a triple $(\mathcal{L},P,(\xi,\mathcal{J}))$ (with two morphological components and one semantico-morphological component) such that:*

- *$\mathcal{L}$ is an ASM vocabulary as in Definition 2.2,*
- *P is an $\mathcal{L}$-program as in Definition 2.7,*
- *ξ is an $\mathcal{L}$-initialization map and $\mathcal{J}$ is a ξ-initial $\mathcal{L}$-state as in Definition 2.5.*

An ASM has type 0 *if all its dynamic symbols have arity* 0 *(i.e., they are constants).*

2. The run of an ASM $(\mathcal{L}, P, (\xi, \mathcal{J}))$ *is the sequence of states* $(\mathcal{S}_i)_{i \in I}$ *indexed by a finite or infinite initial segment* I *of* $\mathbb{N}$ *which is uniquely defined by the following conditions:*

- $\mathcal{S}_0$ *is* $\mathcal{J}$*.*
- $i + 1 \in I$ *if and only if* P *does not clash nor fail nor halt on* $\mathcal{S}_i$ *and* `Active` $(\mathcal{S}_i, P) \neq \emptyset$ *(i.e. there is an active update*[3]*).*
- *If* $i + 1 \in I$ *then* $\mathcal{S}_{i+1} = \mathtt{Succ}(\mathcal{S}_i, P)$*.*

3. Suppose I *is finite and* i *is the maximum element of* I*.*
The run is successful if `Active` $(\mathcal{S}_i, P)$ *is empty or* P *halts on* $\mathcal{S}_i$*. In that case the outputs are the interpretations on* $\mathcal{S}_i$ *of the output symbols.*
The run fails if P *clashes or fails on* $\mathcal{S}_i$*. In that case the run has no output.*

Remark 2.15. In case `Active` $(\mathcal{S}_i, P) \neq \emptyset$ and P does not clash nor fail nor halt on $\mathcal{S}_i$ and $\mathcal{S}_i = \mathcal{S}_{i+1}$ (i.e., if the active updates do not modify $\mathcal{S}_i$) then the run is infinite: $\mathcal{S}_j = \mathcal{S}_i$ for every $j > i$.

2.9 Operational Completeness: the ASM Theorem

Let us now state the fundamental theorem of ASMs.

Theorem 2.16 (ASM Theorem, 1999 [16, 17], cf. [10])**.** *Every process satisfying the Sequential Postulates (cf. §2.2) can be emulated by an ASM with the same vocabulary, sets of states and initial states.*

In other words, using Gurevich Sequential Thesis 2.2, every sequential algorithm can be step by step emulated by an ASM with the same values of all environment parameters. I.e., ASMs are operationally complete as concerns sequential algorithms.

The proof of the ASM Theorem also shows that ASM programs of a remarkably simple form are sufficient.

[3]Nevertheless, it is possible that $\mathcal{S}_i$ and $\mathtt{Succ}(\mathcal{S}_i, P)$ coincide, cf. Remark 2.15.

Definition 2.17. *Let $\mathcal{L}$ be an ASM vocabulary. Two ASM $\mathcal{L}$-programs P, Q are equivalent if, for every $\mathcal{L}$-initialization map ξ and every ξ-initial state $\mathcal{J}$, the two ASMs $(\mathcal{L}, P, (\xi, \mathcal{J}))$ and $(\mathcal{L}, Q, (\xi, \mathcal{J}))$ have exactly the same runs.*

Theorem 2.18 (Gurevich, 1999 [16])**.** *Every ASM program is equivalent to a program which is a parallel block of conditional blocks of updates, halt or fail instructions, namely a program of the form:*

$$\left|\begin{array}{l} \texttt{if } C_1 \texttt{ then } \left|\begin{array}{l} I_{1,1} \\ \vdots \\ I_{1,p_1}\end{array}\right. \\ \vdots \\ \texttt{if } C_n \texttt{ then } \left|\begin{array}{l} I_{n,1} \\ \vdots \\ I_{n,p_n}\end{array}\right. \end{array}\right.$$

where the $I_{i,j}$'s are updates or `Halt` *or* `Fail` *and the interpretations of $C_1, \ldots, C_n$ in any state are Booleans such that at most one of them is* `True`*.*

Proof. For `Skip`, `Halt`, `Fail` consider an empty parallel block. For an update or `Halt` or `Fail` consider a block of one conditional with a tautological condition. Simple Boolean conjunctions allow to transform a conditional of two programs of the wanted form into the wanted form. The same for parallel blocks of such programs. ☐

3 Lambda Calculus

As much as possible, our notations are taken from Barendregt's book [3] (which is a standard reference on Λ-calculus).

3.1 Lambda Terms

Recall that the family Λ of λ-terms of the Λ-calculus is constructed from an infinite family of variables via the following rules:

1. Any variable is a λ-term.
2. *(Abstraction)* If x is a variable and M is a λ-term then $\lambda x \;.\; M$ is a λ-term.

Decorated rules of reduction in Λ-calculus		
(Id) $M \to_0 M$	$(\lambda x.M)\ N \to_1 M[N/x]$	(β)
(App) $\dfrac{M \to_i M'}{MN \to_i M'N}$ $\dfrac{N \to_i N'}{MN \to_i MN'}$	$\dfrac{M \to_i M'}{(\lambda x.M) \to_i \lambda x.M'}$	(Abs)

Figure 2: Reductions with decorations

3. *(Application)* If M, N are λ-terms then $(M\ N)$ is a λ-term.

Free and bound occurrences of a variable in a λ-term are defined as in logical formulas, considering that abstraction $\lambda x\ .\ M$ bounds x in M.

One considers λ-terms up to a renaming (called α-conversion) of their bound variables. In particular, one can always suppose that, within a λ-term, no variable has both free occurrences and bound occurrences and that any two abstractions involve distinct variables.

To simplify notations, it is usual to remove parentheses in terms, according to the following conventions:

- applications associate leftwards: in place of $(\cdots((N_1\ N_2)\ N_3)\cdots\ N_k)$ we write $N_1 N_2 N_3 \cdots N_k$,
- abstractions associate rightwards: $\lambda x_1\ .\ (\lambda x_2\ .\ (\cdots\ .\ (\lambda x_k.M)\cdots))$ is written $\lambda x_1 \cdots x_k\ .\ M$.

3.2 β-Reduction

Note 3.1. Symbols := are used for updates in ASMs and are also commonly used in Λ-calculus to denote by $M[x := N]$ the substitution of all occurrences of a variable x in a term M by a term N. To avoid any confusion, we shall rather denote such a substitution by $M[N/x]$.

The family of λ-terms is endowed with a *reducibility relation*, called β-reduction and denoted by $\to$.

Definition 3.2. *1. Let P be a λ-term. A subterm of P the form $(\lambda x.M)N$ is called a β-redex (or simply redex) of P. Going from P to the λ-term*

Q obtained by substituting in P this redex by $M[N/x]$ (i.e., substituting N to every free occurrence of x in M) is called a β-reduction and we write $P \to Q$.

2. The iterations $\to_i$ of $\to$ and the reflexive and transitive closure $\twoheadrightarrow$ are defined as follows:

$$\begin{array}{lcl} \to_0 & = & \{(M, M) \mid M\} \\ \to_{i+1} & = & \to_i \circ \to \qquad \textit{(so that } \to\ =\ \to_1\textit{)} \\ & = & \{(M_0, M_i) \mid \exists M_1, \ldots, M_i \mid M_0 \to M_1 \to \cdots \to M_i \to M_{i+1}\} \\ \twoheadrightarrow & = & \bigcup_{i \in \mathbb{N}} \to_i \end{array}$$

These reduction relations are conveniently expressed via axioms and rules (cf. Figure 1): the schema of axioms (β) gives the core transformation whereas rules (App) and (Abs) insure that this can be done for subterms.

Relations $\to_i$ are of particular interest to analyse the complexity of the simulation of one ASM step in Λ-calculus. Observe that axioms and rules for $\to$ extend to $\twoheadrightarrow$.

3.3 Normal Forms

Definition 3.3. *1. A λ-term M is in normal form if it contains no redex.*

2. A λ-term M has a normal form if there exists some term N in normal form such that $M \twoheadrightarrow N$.

Remark 3.4. There are terms with no normal form. The classical example is $\Omega = \Delta\Delta$ where $\Delta = \lambda x\ .\ xx$. Indeed, Ω is a redex and reduces to itself.

In a λ-term, there can be several subterms which are redexes, so that iterating $\to$ reductions is a highly non deterministic process. Nevertheless, going to normal form is a functional process.

Theorem 3.5 (Church-Rosser [7], 1936)**.** *The relation $\twoheadrightarrow$ is confluent: if $M \twoheadrightarrow N'$ and $M \twoheadrightarrow N''$ then there exists P such that $N' \twoheadrightarrow P$ and $N'' \twoheadrightarrow P$. In particular, there exists at most one term N in normal form such that $M \twoheadrightarrow N$.*

Remark 3.6. Theorem 3.5 deals with $\twoheadrightarrow$ exclusively: relation $\to_i$ is *not* confluent for any $i \geq 1$.

A second fundamental property is that going to normal form can be made a deterministic process.

Definition 3.7. *Let R', R'' be two occurrences of redexes in a term P. We say that R' is left to R'' if the first lambda in R' is left to the first lambda in R'' (all this viewed in P). If terms are seen as labelled ordered trees, this means that the top lambda in R' is smaller than that in R'' relative to the prefix ordering on nodes of the tree P.*

Theorem 3.8 (Curry & Feys [9], 1958)**.** *Reducing the leftmost redex of terms not in normal form is a deterministic strategy which leads to the normal form if there is some.*
In other words, if M has a normal form N then the sequence $M = M_0 \to M_1 \to M_2 \to \cdots$ where each reduction $M_i \to M_{i+1}$ reduces the leftmost redex in M_i (if M_i is not in normal form) is necessarily finite and ends with N.

3.4 Lists in Λ-Calculus

We recall the usual representation of lists in Λ-calculus with special attention to decoration (i.e., the number of β-reductions in sequences of reductions).

Proposition 3.9. *Let $\langle u_1, \ldots, u_k\rangle = \lambda z \, . \, z u_1 \ldots u_k$ and, for $i = 1, \ldots, k$, let $\pi_i^k = \lambda x_1 \ldots x_k \, . \, x_i$. Then $\langle u_1, \ldots, u_k\rangle \; \pi_i^k \to_{1+k} \; u_i$.*
Moreover, if all u_i's are in normal form then so is $\langle u_1, \ldots, u_k\rangle$ and these reductions are deterministic: there exists a unique sequence of reductions from $\langle u_1, \ldots, u_k\rangle$ to u_i.

3.5 Booleans in Λ-Calculus

We recall the usual representation of Booleans in Λ-calculus.

Proposition 3.10. *Boolean elements* `True`, `False` *and usual Boolean functions can be represented by the following λ-terms, all in normal form:*

			neg	$=$	$\lambda x \, . \, x \ulcorner\texttt{False}\urcorner \ulcorner\texttt{True}\urcorner$
$\ulcorner\texttt{True}\urcorner$	$=$	$\lambda xy.x$	and	$=$	$\lambda xy \, . \, xy \ulcorner\texttt{False}\urcorner$
$\ulcorner\texttt{False}\urcorner$	$=$	$\lambda xy.y$	or	$=$	$\lambda xy \, . \, x \ulcorner\texttt{True}\urcorner \, y$
			implies	$=$	$\lambda xy \, . \, xy \ulcorner\texttt{True}\urcorner$
			iff	$=$	$\lambda xy \, . \, xy(\ulcorner\neg\urcorner y)$

For $a, b \in \{\texttt{True}, \texttt{False}\}$, we have neg $\ulcorner a\urcorner \to \ulcorner\neg a\urcorner$, and $\ulcorner a\urcorner \ulcorner b\urcorner \twoheadrightarrow \ulcorner a f b\urcorner, \ldots$.

Proposition 3.11 (If Then Else)**.** *For all terms M, N,*

$$(\lambda z\ .\ zMN)\ \ulcorner\mathtt{True}\urcorner \rightarrow_2 M \quad , \quad (\lambda z\ .\ zMN)\ \ulcorner\mathtt{False}\urcorner \rightarrow_2 N\ .$$

We shall use the following version of iterated conditional.

Proposition 3.12. *For every $n \geq 1$ there exists a term $Case_n$ such that, for all normal terms $M_1, \ldots, M_n$ and all $t_1, \ldots, t_n \in \{\ulcorner\mathtt{True}\urcorner, \ulcorner\mathtt{False}\urcorner\}$,*

$$Case_n\ M_1 \ldots M_n\ t_1 \ldots t_n \rightarrow_{3n} M_i$$

relative to leftmost reduction in case $t_i = \ulcorner\mathtt{True}\urcorner$ and $\forall j < i\ t_j = \ulcorner\mathtt{False}\urcorner$.

Proof. Let $u_i = y_i(\lambda x_{i+1}\ .\ I) \ldots (\lambda x_n\ .\ I)$, set

$$Case_n \quad = \quad \lambda y_1 \ldots y_n z_1 \ldots z_n\ .\ z_1 u_1(z_2 u_2(\ldots(z_{n-1}u_{n-1}(z_n u_n I))\ldots))$$

and observe that, for leftmost reduction, letting $M'_i = u_i[M_i/y_i]$,

$$\begin{array}{rcl} Case_n\ M_1 \ldots M_n\ t_1 \ldots t_n & \rightarrow_{2n} & t_1 M'_1(t_2 M'_2(\ldots(t_{n-1}M'_{n-1}(t_n M'_n I))\ldots)) \\ & \rightarrow_i & M'_i \\ & \rightarrow_{n-i} & M_i\ . \end{array}$$

□

3.6 Integers in Λ-Calculus

There are several common representations of integers in Λ-calculus. We shall consider a slight variant of the standard one (we choose another term for $\ulcorner 0\urcorner$), again with special attention to decoration.

Proposition 3.13. *Let*

$$\boxed{\begin{array}{rcl} \ulcorner 0\urcorner & = & \lambda z\ .\ z\ulcorner\mathtt{True}\urcorner\ulcorner\mathtt{False}\urcorner \\ \ulcorner n+1\urcorner & = & \langle\ulcorner\mathtt{False}\urcorner, \ulcorner n\urcorner\rangle \qquad = \quad \lambda z\ .\ z\ulcorner\mathtt{False}\urcorner\ulcorner n\urcorner \\ Zero & = & \lambda x\ .\ x\ulcorner\mathtt{True}\urcorner \\ Succ & = & \lambda z\ .\ \langle\ulcorner\mathtt{False}\urcorner, z\rangle \\ Pred & = & \lambda z\ .\ x\ulcorner\mathtt{False}\urcorner \end{array}}$$

The above terms are all in normal form and

$$\begin{array}{rcl} Zero\ulcorner 0\urcorner & \rightarrow_3 & \ulcorner\mathtt{True}\urcorner \\ Zero\ulcorner n+1\urcorner & \rightarrow_3 & \ulcorner\mathtt{False}\urcorner \end{array} \quad , \quad \begin{array}{rcl} Succ\ulcorner n\urcorner & \rightarrow_3 & \ulcorner n+1\urcorner \\ Pred\ulcorner n+1\urcorner & \rightarrow_3 & \ulcorner n\urcorner \\ Pred\ulcorner 0\urcorner & \rightarrow_3 & \ulcorner\mathtt{False}\urcorner \end{array} \quad .$$

Moreover, all these reductions are deterministic.

Remark 3.14. The standard definition sets $\ulcorner 0 \urcorner = \lambda x \,.\, x$. Observe that $\mathit{Zero}(\lambda x \,.\, x) \to_2 \ulcorner \mathtt{True} \urcorner$. The chosen variant of $\ulcorner 0 \urcorner$ is to get the same decoration (namely 3) to go from $\mathit{Zero}\ulcorner 0 \urcorner$ to $\ulcorner \mathtt{True} \urcorner$ and to go from $\mathit{Zero}\ulcorner n+1 \urcorner$ to $\ulcorner \mathtt{False} \urcorner$.

Let us recall Kleene's fundamental result.

Theorem 3.15 (Kleene, 1936)**.** *For every partial computable function $f : \mathbb{N}^k \to \mathbb{N}$ there exists a λ-term M such that, for every tuple $(n_1, \cdots, n_k)$,*

- *$M\ulcorner n_1 \urcorner \cdots \ulcorner n_k \urcorner$ admits a normal form (i.e., is $\twoheadrightarrow$ reducible to a term in normal form) if and only if $(n_1, \cdots, n_k)$ is in the domain of f,*
- *in that case, $M\ulcorner n_1 \urcorner \cdots \ulcorner n_k \urcorner \twoheadrightarrow \ulcorner f(n_1, \cdots, n_k) \urcorner$ (and, by Theorem 3.5, this normal form is unique).*

3.7 Datatypes in Λ-Calculus

We just recalled some representations of Booleans and integers in Λ-calculus. In fact, any inductive datatype can also be represented. Using computable quotienting, this allows to also represent any datatype used in algorithms. Though we will not extend on this topic, let us recall Scott encoding of inductive datatypes in the Λ-calculus (cf. Mogensen [23]).

> *1. If the inductive datatype has constructors $\psi_1, \ldots, \psi_p$ having arities $k_1, \ldots, k_p$, constructor ψ_i is represented by the term*
>
> $$\lambda x_1 \ldots x_{k_i} \alpha_1 \ldots \alpha_p \,.\, \alpha_i x_1 \ldots x_{k_i} \,.$$
>
> *In particular, if ψ_i is a generator (i.e., an arity 0 constructor) then it is represented by the projection term $\lambda \alpha_1 \ldots \alpha_p \,.\, \alpha_i$.*
> *2. An element of the inductive datatype is a composition of the constructors and is represented by the similar composition of the associated λ-terms.*

Extending the notations used for Booleans and integers, we shall also denote by $\ulcorner a \urcorner$ the λ-term representing an element a of a datatype.

Scott's representation of inductive datatypes extends to finite families of datatypes defined via mutual inductive definitions. It suffices to endow constructors with types and to restrict compositions in point 2 above to those respecting constructor types.

3.8 Lambda Calculus with Benign Constants

We consider an extension of the lambda calculus with constants to represent particular computable functions and predicates. Contrary to many $\lambda\delta$-calculi (Church $\lambda\delta$-calculus, 1941 [8], Statman, 2000 [26], Ronchi Della Rocca, 2004 [25], Barendregt & Statman, 2005 [4]), this adds no real additional power: it essentially allows for shortcuts in sequences of reductions. The reason is that axioms in Definition 3.16 do not apply to all terms but only to codes of elements in datatypes.

Definition 3.16. *Let $\mathbb{F}$ be a family of functions with any arities over some datatypes $A_1, \ldots, A_n$. The $\Lambda_{\mathbb{F}}$-calculus is defined as follows:*

- *The family of $\lambda_{\mathbb{F}}$-terms is constructed as in §3.1 from the family of variables augmented with constant symbols: one constant c_f for each $f \in \mathbb{F}$.*
- *The axioms and rules of the top table of Figure 2 are augmented with the following axioms: if $f : A_{i_1} \times \cdots \times A_{i_k} \to A_i$ is in $\mathbb{F}$ then, for all $(a_1, \cdots, a_k) \in A_{i_1} \times \cdots \times A_{i_k}$,*

$$(Ax_f) \qquad\qquad c_f \ulcorner a_1 \urcorner \cdots \ulcorner a_k \urcorner \ \to\ \ulcorner f(a_1, \cdots, a_k) \urcorner\ .$$

Definition 3.17. *1. We denote by $\to_\beta$ the classical β-reduction (with the contextual rules (Abs), (App)) extended to terms of $\Lambda_{\mathbb{F}}$.*
2. We denote by $\to_{\mathbb{F}}$ the reduction given by the sole (Ax_f)-axioms and the contextual rules (Abs), (App).
3. We use double decorations: $M \ \to_{i,j} N$ means that there is a sequence consisting of i β-reductions and j $\mathbb{F}$-reductions which goes from t to u.

The Church-Rosser property still holds.

Proposition 3.18. *The $\Lambda_{\mathbb{F}}$-calculus is confluent (cf. Theorem 3.5).*

Proof. Theorem 3.5 insures that $\twoheadrightarrow_\beta$ is confluent. It is immediate to see that any two applications of the $\mathbb{F}$ axioms can be permuted: this is because two distinct $\mathbb{F}$-redexes in a term are always disjoint subterms. Hence $\to_{\mathbb{F}}$ is confluent. Observe that $\twoheadrightarrow$ is obtained by iterating finitely many times the relation $\twoheadrightarrow_\beta \cup \to_{\mathbb{F}}$. Using Hindley-Rosen Lemma (cf. Barendregt's book [3], Proposition 3.3.5, or Hankin's book [19], Lemma 3.27), to prove that $\twoheadrightarrow$ is confluent, it suffices to prove that $\twoheadrightarrow_\beta$ and $\to_\mathbb{F}$ commute. One easily reduces to prove that $\to_\beta$ and $\to_\mathbb{F}$ commute, i.e.,

$$\exists P\ (M\ \to_\beta P\ \to_\mathbb{F} N) \quad\iff\quad \exists Q\ (M\ \to_\mathbb{F} Q\ \to_\beta N)\ .$$

Any length two such sequence of reductions involves two redexes in the term M: a β-redex $R = (\lambda x\ .\ A)B$ and a $\mathbb{F}$-redex $C = c\ \ulcorner a_1 \urcorner \cdots \ulcorner a_k \urcorner$. There are three cases: either R and C are disjoint subterms of M or C is a subterm of A or C is a subterm of B. Each of these cases is straightforward. $\square$

We adapt the notion of leftmost reduction in the $\Lambda_\mathbb{F}$-calculus as follows.

Definition 3.19. *The leftmost reduction in $\Lambda_\mathbb{F}$ reduces the leftmost $\mathbb{F}$-redex if there is some else it reduces the leftmost β-redex.*

3.9 Good $\mathbb{F}$-Terms

To functions which can be obtained by composition from functions in $\mathbb{F}$ we associate canonical terms in $\Lambda_\mathbb{F}$ and datatypes. These canonical terms are called good $\mathbb{F}$-terms, they contain no abstraction, only constant symbols c_f, with $f \in \mathbb{F}$, and variables.

Problem 3.20. We face a small problem. Functions in $\mathbb{F}$ are to represent static functions of an ASM. Such functions are typed whereas $\Lambda_\mathbb{F}$ is an untyped lambda calculus. In order to respect types when dealing with composition of functions in $\mathbb{F}$, the definition of good $\mathbb{F}$-terms is done in two steps: the first step involves typed variables and the second one replaces them by untyped variables.

Definition 3.21. *1. Let $A_1, \ldots, A_n$ be the datatypes involved in functions of the family $\mathbb{F}$. Consider typed variables $x_j^{A_i}$ where $j \in \mathbb{N}$ and $i = 1, \ldots, n$. The family of pattern $\mathbb{F}$-terms, their types and semantics are defined as follows: Let $f \in \mathbb{F}$ be such that $f : A_{i_1} \times \cdots \times A_{i_k} \to A_q$.*

- *If $x_{j_1}^{A_{i_1}}, \ldots, x_{j_k}^{A_{i_k}}$ are typed variables then the term $c_f\ x_{j_1}^{A_{i_1}} \ldots x_{j_k}^{A_{i_k}}$ is a pattern $\mathbb{F}$-term with type $A_{i_1} \times \cdots \times A_{i_k} \to A_q$ and semantics $[\![c_f\ x_{j_1}^{A_{i_1}} \ldots x_{j_k}^{A_{i_k}}]\!] = f$.*
- *For $j = 1, \ldots, k$, let t_j be a pattern $\mathbb{F}$-term with datatype A_j or a typed variable $x_i^{A_j}$. Suppose the term $t = c_f\ t_1 \cdots t_k$ contains exactly the typed*

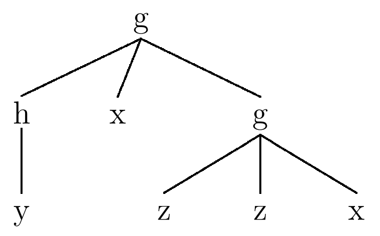

Figure 3: Composition tree

variables $x_j^{A_i}$ for $(i,j) \in I$ and, for $\ell = 1, \ldots, k$, the term t_ℓ contains exactly the typed variables $x_j^{A_i}$ for $(i,j) \in I_j \subseteq I$.
Then the term $c_f\ t_1 \cdots t_k$ is a pattern $\mathbb{F}$-term with type $\prod_{i \in I} A_i \to A_q$ and a semantics $[\![c_f\ t_1 \cdots t_k]\!]$ such that, for every tuple $(a_i)_{i \in I} \in \prod_{i \in I} A_i$,

$$[\![t]\!]((a_i)_{i \in I}) = f([\![t_1]\!]((a_i)_{i \in I_1})), \ldots, [\![t_k]\!]((a_k)_{i \in I_k}))) \ .$$

2. Good $\mathbb{F}$-terms are obtained by substituting in a pattern $\mathbb{F}$-term untyped variables to the typed variables so that two distinct typed variables are substituted by two distinct untyped variables.

The semantics of good $\mathbb{F}$-terms is best illustrated by the following example: the function h associated to the term $c_g(c_h y)x(c_g zzx)$ is the one given by equality $f(x,y,z) = g(h(y), x, g(z,z,x))$ which corresponds to Figure 3.9.

The reason for the above definition is the following simple result about reductions of good terms obtained via substitutions. It is proved via a straightforward induction on good $\mathbb{F}$-terms and will be used in §4.3, 4.4

Proposition 3.22. *Let t be a good $\mathbb{F}$-term with k variables $y_1, \ldots, y_k$ such that $[\![t]\!] = f : A_{i_1} \times \cdots \times A_{i_k} \to A_q$. Let N be the number of nodes of the tree associated to the composition of functions in $\mathbb{F}$ giving f (cf. Figure 3.9). There exists $L_t = O(N)$ such that, for every $(a_1, \ldots, a_k) \in A_{i_1} \times \cdots \times A_{i_k}$,*

$$t[\ulcorner a_1 \urcorner / y_1, \ldots, \ulcorner a_k \urcorner / y_k] \twoheadrightarrow_{\mathbb{F}} \ulcorner f(a_1, \ldots, a_k) \urcorner$$

and, using the leftmost reduction strategy, this sequence of reductions consists of exactly L_t $\mathbb{F}$-reductions.

4 Variations on Curry's Fixed Point

4.1 Curry's Fixed Point

Let us recall Curry's fixed point.

Definition 4.1. *The Curry operator $\varphi \mapsto \theta_\varphi$ on λ-terms is defined as follows*

$$\theta_F = (\lambda x \ . \ F(xx))(\lambda x \ . \ F(xx)) \ .$$

Theorem 4.2 (Curry's fixed point)**.** *For every λ-term F, $\theta_F \to F\theta_F$.*

Proof. One β-reduction suffices: θ_F is of the form XX and is itself a redex (since X is an abstraction) which β-reduces to $F(XX)$, i.e., to $F\theta_F$. $\square$

4.2 Padding Reductions

We show how to pad leftmost reduction sequences so as to get prescribed numbers of β and $\mathbb{F}$-reductions.

Lemma 4.3 (Padding lemma)**.** *Suppose that $\mathbb{F}$ contains some function $\omega : B_1 \times \cdots \times B_\ell \to B_i$ (with $1 \leq i \leq \ell$) and some constants $\nu_1 \in B_1, \ldots, \nu_\ell \in B_\ell$.*
1. For every $K \geq 2$ and $L \geq 0$, there exists a λ-term $pad_{K,L}$ in $\Lambda_\mathbb{F}$ with length $O(K+L)$ such that, for any finite sequence of λ-terms $\theta, t_1, \ldots, t_k$ in $\Lambda_\mathbb{F}$ which contain no $\mathbb{F}$-redex,

i. $pad_{K,L}\ \theta\ t_1 \cdots t_k \ \twoheadrightarrow\ \theta\ t_1 \cdots t_k$.

ii. The leftmost derivation consists of exactly L $\mathbb{F}$-reductions followed by K β-reductions.

2. Moreover, if $K \geq 3$, one can also suppose that $pad_{K,L}$ contains no $\mathbb{F}$-redex.

Proof. 1. For the sake of simplicity, we suppose that ω has arity 1, the general case being a straightforward extension. Let $I = \lambda x \ . \ x$ and $I^\ell = I \cdots I$ (ℓ times I). Observe that $I^\ell\ s_0 \cdots s_p \ \twoheadrightarrow\ s_0 \cdots s_p$ and the leftmost derivation consists of exactly ℓ β-reductions. So it suffices to set $pad_{K,0} = I^K$ and, for $L \geq 1$,

$$pad_{K,L} = I^{K-2}\ (\lambda xy \ . \ y)\ (\overbrace{\ulcorner\omega\urcorner(\ldots(\ulcorner\omega\urcorner}^{L \text{ times}}\ulcorner\nu_1\urcorner)\ldots)) \ .$$

2. To suppress the $\mathbb{F}$-redex $\ulcorner\omega\urcorner\ulcorner\nu_1\urcorner$, modify $pad_{K,L}$ as follows:

$$pad_{K,L} = I^{K-3}\ (\lambda xy\ .\ xy)\ ((\lambda z\ .\ \overbrace{(\ulcorner\omega\urcorner(\ldots(\ulcorner\omega\urcorner}^{L \text{ times}} z)\ldots)))\ \ulcorner\nu_1\urcorner)\ .$$

□

4.3 Constant Cost Updates

We use Curry's fixed point Theorem and the above padding technique to insure constant length reductions for any given update function for tuples.

Lemma 4.4. *Let $A_1,\ldots,A_n$ be the datatypes involved in functions of the family $\mathbb{F}$. Suppose that $\mathbb{F}$ contains some function $\omega : B_1\times\cdots\times B_\ell \to B_i$ (with $1\le i\le \ell$) and some constants $\nu_1\in B_1$, ..., $\nu_\ell\in B_\ell$. Let $\tau:\{1,\ldots,k\}\to\{1,\ldots,n\}$ be a distribution of indexes of sorts. For $j=1,\ldots,k$, let φ_j be a good $\mathbb{F}$-term with variables x_i for $i\in I_j\subseteq\{1,\ldots,k\}$ such that $[\![\varphi_j]\!] = f_j : \prod_{i\in I_j} A_{\tau(i)}\to A_{\tau(j)}$.*
There exists constants K_{min} and L_{min} such that, for all $K\ge K_{min}$ and $L\ge L_{min}$, there exists a λ-term θ such that,

1. *Using the leftmost reduction strategy, for all $(a_1,\ldots,a_k)\in A_{\tau(i)}\times\cdots\times A_{\tau(k)}$, denoting by $\vec{a}_I$ the tuple $(a_j)_{j\in I}$,*
$$\theta\ \ulcorner a_1\urcorner\cdots\ulcorner a_k\urcorner\ \twoheadrightarrow\ \theta\ \ulcorner f_1(\vec{a}_{I_1})\urcorner\cdots\ulcorner f_k(\vec{a}_{I_k})\urcorner\ . \qquad (1)$$
2. *This sequence of reductions consists of K β-reductions and L $\mathbb{F}$-reductions.*

Proof. Let K', L' be integers to be fixed later on. Set

$$F = pad_{K',L'}\ \lambda\alpha x_1\ldots x_k\ .\ \alpha\varphi_1\ldots\varphi_k \qquad \theta = (\lambda z\ .\ F(zz))\ (\lambda z\ .\ F(zz))\ .$$

Since θ and the φ_i's have no $\mathbb{F}$-redex, we have the following leftmost reduction:

$$\begin{array}{lll}
\theta\ \ulcorner a_1\urcorner\cdots\ulcorner a_k\urcorner & \to_{1,0} & F\ \theta\ \ulcorner a_1\urcorner\cdots\ulcorner a_k\urcorner \quad \text{(cf. Theorem 4.2)}\\
 & = & pad_{K',L'}\ (\lambda\alpha x_1\ldots x_k\ .\ \alpha\varphi_1\ldots\varphi_k)\ \theta\ \ulcorner a_1\urcorner\cdots\ulcorner a_k\urcorner\\
 & \to_{K',L'} & (\lambda\alpha x_1\ldots x_k\ .\ \alpha\varphi_1\ldots\varphi_k)\ \theta\ \ulcorner a_1\urcorner\cdots\ulcorner a_k\urcorner\\
 & & \quad \text{(apply Lemma 4.3)}\\
 & \to_{k+1,0} & \theta\ \varphi_1[\ulcorner a_1\urcorner/x_1,\ldots,\ulcorner a_k\urcorner/x_k]\\
 & & \qquad\qquad \cdots\varphi_k[\ulcorner a_1\urcorner/x_1,\ldots,\ulcorner a_k\urcorner/x_k]\\
 & \to_{0,S} & \theta\ \ulcorner f_1(\vec{a}_{I_1})\urcorner\cdots\ulcorner f_k(\vec{a}_{I_k})\urcorner\\
 & & \quad \text{(apply Proposition 3.22)}
\end{array}$$

where $S = \sum_{j=1,\dots,k} L_{\varphi_j}$. The total cost is $K'+k+2$ β-reductions plus $L'+S$ $\mathbb{F}$-reductions. We conclude by setting $K' = K-(k+2)$ and $L' = L-S$. $\square$

4.4 Constant Cost Conditional Updates

We refine Lemma 4.4 to conditional updates.

Lemma 4.5. *Let $A_1,\dots,A_n$ be the datatypes involved in functions of the family $\mathbb{F}$. Suppose that $\mathbb{F}$ contains some function $\omega : B_1 \times \cdots \times B_\ell \to B_i$ (with $1 \le i \le \ell$) and some constants $\nu_1 \in B_1, \dots, \nu_\ell \in B_\ell$. Let $\tau : \{1,\dots,k\} \to \{1,\dots,n\}$, $\iota_1,\dots,\iota_q \in \{1,\dots,n\}$ be distributions of indexes of sorts. Let $(\rho_s)_{s=1,\dots,p+q}$, $(\varphi_{i,j})_{i=1,\dots,p,j=1,\dots,k}$, $(\gamma_\ell)_{i=1,\dots,q}$ be sequences of good $\mathbb{F}$-terms with variables x_i with i varying in the respective sets $I_s, I_{i,j}, J_\ell \subseteq \{1,\dots,k\}$. Suppose that*

$$\begin{array}{rcccccl} [\![\rho_s]\!] &=& r_s &:& \prod_{i\in I_s} A_{\tau(i)} &\to& \texttt{Bool} \ , \\ [\![\varphi_{i,j}]\!] &=& f_{i,j} &:& \prod_{i\in I_{i,j}} A_{\tau(i)} &\to& A_{\tau(j)} \ , \\ [\![\gamma_\ell]\!] &=& g_\ell &:& \prod_{i\in J_\ell} A_{\tau(i)} &\to& A_{\iota(\ell)} \end{array}$$

(in particular, $f_{1,j},\dots,f_{p,j}$ all take values in $A_{\tau(j)}$). There exists constants K_{min} and L_{min} such that, for all $K \ge K_{min}$ and $L \ge L_{min}$, there exists a λ-term θ such that,

1. *Using the leftmost reduction strategy, for all $(a_1,\dots,a_k) \in A_{\tau(1)} \times \cdots \times A_{\tau(k)}$ and $s \in \{1,\dots,p,p+1,\dots,p+q\}$,*

$$\text{If} \qquad r_s(\vec{a}_{I_s}) = \texttt{True} \ \wedge \ \forall t < s \ r_t(\vec{a}_{I_t}) = \texttt{False} \qquad\qquad (\dagger)_s$$

$$\text{then} \quad \theta \ulcorner a_1 \urcorner \cdots \ulcorner a_k \urcorner \ \twoheadrightarrow \ \begin{cases} \theta \ulcorner f_{s,1}(\vec{a}_{I_{s,1}}) \urcorner \cdots \ulcorner f_{s,k}(\vec{a}_{I_{s,k}}) \urcorner & \text{if } s \le p \\ \ulcorner g_\ell(\vec{a}_{J_\ell}) \urcorner & \text{if } s = p+\ell \end{cases} .$$

2. *In all cases, this sequence of reductions consists of exactly K β-reductions and L $\mathbb{F}$-reductions.*

Proof. Let K', L' be integers to be fixed at the end of the proof. For $i = 1,\dots,p$ and $\ell = 1,\dots,q$, let

$$M_i = \alpha\varphi_{i,1}\cdots\varphi_{i,k} \qquad M_{p+\ell} = \gamma_\ell \ .$$

Using the $Case_n$ term from Proposition 3.12, set

$$\begin{array}{rcl} H & = & Case_{p+q} M_1 \ldots M_p M_{p+1} \ldots M_{p+q} \\ G & = & \lambda \alpha x_1 \ldots x_k \;.\; (H\ \rho_1(x_1 \ldots x_k) \ldots \rho_{p+q}(x_1 \ldots x_k)) \\ F & = & pad_{K',L'}\ G \\ \theta & = & (\lambda z \;.\; F(zz))\ (\lambda z \;.\; F(zz)) \end{array}$$

The following sequence of reductions is leftmost because, as long as $pad_{K',L'}$ is not completely reduced, there is no $\mathbb{F}$-redex on its right.

$$(R_1) \quad \begin{array}{rcll} \theta \ulcorner a_1 \urcorner \cdots \ulcorner a_k \urcorner & \rightarrow_{1,0} & F\ \theta \ulcorner a_1 \urcorner \cdots \ulcorner a_k \urcorner & \text{(cf. Theorem 4.2)} \\ & = & pad_{K',L'}\ G\ \theta \ulcorner a_1 \urcorner \cdots \ulcorner a_k \urcorner & \\ & \rightarrow_{K',L'} & G\ \theta \ulcorner a_1 \urcorner \cdots \ulcorner a_k \urcorner & \end{array}$$

Let us denote by A^σ the term $A^\sigma_i = A[\theta/\alpha, \ulcorner a_1 \urcorner/x_{,1}, \ldots, \ulcorner a_k \urcorner/x_k]$. The leftmost reduction sequence goes on with β-reductions as follows:

$$(R_2) \quad \begin{array}{rcl} G\ \theta \ulcorner a_1 \urcorner \cdots \ulcorner a_k \urcorner & = & (\lambda \alpha x_1 \ldots x_k \;.\; (H\ \rho_1 \ldots \rho_{p+q}))\ \theta \ulcorner a_1 \urcorner \cdots \ulcorner a_k \urcorner \\ & \rightarrow_{k+1,0} & H^\sigma\ \rho^\sigma_1 \ldots \rho^\sigma_{p+q} \end{array}$$

Now, using Proposition 3.22, the following leftmost reductions are $\mathbb{F}$-reductions:

$$\begin{array}{rcrcl} & & \varphi^\sigma_{i,j} & \rightarrow_{0,L_{\varphi_{i,j}}} & \ulcorner f_{i,j}(\vec{a}_{I_{i,j}}) \urcorner \\ & & M^\sigma_i & \rightarrow_{0,\sum_{j=1}^{j=k} L_{\varphi_{i,j}}} & \theta \ulcorner f_{i,1}(\vec{a}_{I_{i,1}}) \urcorner \ldots \ulcorner f_{i,k}(\vec{a}_{I_{i,k}}) \urcorner \\ M^\sigma_{p+\ell} & = & \gamma^\sigma_{p+\ell} & \rightarrow_{0,L_{\gamma_\ell}} & \ulcorner g_\ell(\vec{a}_{J_\ell}) \urcorner \\ & & \rho^\sigma_s & \rightarrow_{0,L_{\rho_s}} & \ulcorner r_s(\vec{a}_{I_s}) \urcorner \end{array}$$

Going with our main leftmost reduction sequence, letting

$$N = (\sum_{i=1}^{i=p} \sum_{j=1}^{j=k} L_{\varphi_{i,j}}) + \sum_{\ell=1}^{\ell=q} L_{\gamma_\ell} + \sum_{s=1}^{s=p+q} L_{\rho_s}$$

and s be as in condition $(\dagger)_s$ in the statement of the Lemma, we get

$$(R_3) \quad \begin{array}{rcl} H^\sigma\ \rho^\sigma_1 \ldots \rho^\sigma_{p+q} & = & Case_{p+q} M^\sigma_1 \ldots M^\sigma_p M^\sigma_{p+1} \ldots M^\sigma_{p+q}\ \rho^\sigma_1 \ldots \rho^\sigma_{p+q} \\ & \rightarrow_{0,N} & Case_{p+q} \\ & & \quad (\theta \ulcorner f_{1,1}(\vec{a}_{I_{1,1}}) \urcorner \ldots \ulcorner f_{1,k}(\vec{a}_{I_{1,k}}) \urcorner) \\ & & \quad \ldots \\ & & \quad (\theta \ulcorner f_{p,1}(\vec{a}_{I_{p,1}}) \urcorner \ldots \ulcorner f_{p,k}(\vec{a}_{I_{p,k}}) \urcorner) \\ & & \quad (\ulcorner g_1(\vec{a}_{J_1}) \urcorner) \quad \ldots \quad (\ulcorner g_q(\vec{a}_{J_q}) \urcorner) \\ & & \quad \rho^\sigma_1 \quad \ldots \quad \rho^\sigma_{p+q} \\ & \rightarrow_{3(p+q),0} & \begin{cases} \theta \ulcorner f_{s,1}(\vec{a}_{I_{s,1}}) \urcorner \ldots \ulcorner f_{s,k}(\vec{a}_{I_{s,k}}) \urcorner & \mathit{if}\ s \le p \\ \ulcorner g_\ell(\vec{a}_{J_\ell}) \urcorner) & \mathit{if}\ s = p + \ell \end{cases} \end{array}$$

Summing up reductions (R_1), (R_2), (R_3), we see that

$$\theta \ulcorner a_1 \urcorner \ldots \ulcorner a_k \urcorner \ \rightarrow_{\eta,\zeta} \ \begin{cases} \theta \ulcorner f_{s,1}(\vec{a}_{I_{s,1}}) \urcorner \ldots \ulcorner f_{s,k}(\vec{a}_{I_{s,k}}) \urcorner & \text{if } s \leq p \\ \ulcorner g_\ell(\vec{a}_{J_\ell}) \urcorner) & \text{if } s = p + \ell \end{cases}$$

where $\eta = 1 + K' + (k+1) + 3(p+q)$ and $\zeta = L' + N$.
To conclude, set $K_{min} = k + 5 + 3(p+q)$ and $L_{min} = N$. If $K \geq K_{min}$ and $L \geq L_{min}$ it suffices to set $K' = K - (K_{min} - 3)$ and $L' = L - L_{min}$ and to observe that $K' \geq 3$ as needed in Lemma 4.3. □

5 ASMs and Lambda Calculus

All along this section, $\mathcal{S} = (\mathcal{L}, P, (\xi, \mathcal{J}))$ is some fixed ASM (cf. Definition 2.14).

5.1 Datatypes and ASM Base Sets

The definition of ASM does not put any constraint on the base sets of the multialgebra. However, only elements which can be named are of any use, i.e. elements which are in the range of compositions of (static or dynamic) functions on the ASM at the successive steps of the run.
The following straightforward result formalizes this observation.

Proposition 5.1. *Let $(\mathcal{L}, P, (\xi, \mathcal{J}))$ be an ASM. Let $\mathcal{U}_1, \ldots, \mathcal{U}_n$ be the base sets interpreting the different sorts of this ASM. For $t \in \mathbb{N}$, let $A_1^{(t)} \subseteq \mathcal{U}_1, \ldots, A_n^{(t)} \subseteq \mathcal{U}_n$ be the sets of values of all ground good $\mathbb{F}$-terms (i.e. with no variable) in the t-th successor state $\mathcal{S}_t$ of the initial state $\mathcal{J}$ of the ASM.*

1. *For any $t \in \mathbb{N}$, $A_1^{(t)} \supseteq A_1^{(t+1)}, \ldots, A_n^{(t)} \supseteq A_n^{(t+1)}$.*

2. *$(A_1^{(t)}, \ldots, A_n^{(t)})$ is a submultialgebra of $\mathcal{S}_t$, i.e. it is closed under all static and dynamic functions of the state $\mathcal{S}_t$.*

Thus, the program really works only on the elements of the sets $(A_1^{(0)}, \ldots, A_n^{(0)})$ of the initial state which are datatypes defined via mutual inductive definitions using ξ and $\mathcal{J}$.

5.2 Tailoring Lambda Calculus for an ASM

Let $\mathbb{F}$ be the family of interpretations of all static symbols in the initial state. The adequate Lambda calculus to encode the ASM is $\Lambda_{\mathbb{F}}$.
Let us argue that this is not an unfair trick. An algorithm does decompose a task in elementary ones. But "elementary" does not mean "trivial" nor "atomic", it just means that we do not detail how they are performed: they are like oracles. There is no absolute notion of elementary task. It depends on what big task is under investigation. For an algorithm about matrix product, multiplication of integers can be seen as elementary. Thus, algorithms go with oracles.
Exactly the same assumption is done with ASMs: static and input functions are used for free.

5.3 Main Theorem for Type 0 ASMs

We first consider the case of type 0 ASMs.

Theorem 5.2. *Let $(\mathcal{L}, P, (\xi, \mathcal{J}))$ be an ASM with base sets $\mathcal{U}_1, \ldots, \mathcal{U}_n$. Let $A_1, \ldots, A_n$ be the datatypes $A_1^{(0)}, \ldots, A_n^{(0)}$ (cf. Proposition 5.1). Let $\mathbb{F}$ be the family of interpretations of all static symbols of the ASM restricted to the datatypes $A_1, \ldots, A_n$. Suppose all dynamic symbols have arity 0, i.e. all are constants symbols. Suppose these dynamic symbols are $\eta_1, \ldots, \eta_k$. and $\eta_1, \ldots, \eta_\ell$ are the output symbols.*
Let us denote by e_i^t the value of the constant η_i in the t-th successor state $\mathcal{S}_t$ of the initial state $\mathcal{J}$.
There exists K_0 such that, for every $K \geq K_0$, there exists a λ-term θ in $\Lambda_{\mathbb{F}}$ such that, for all initial values $e_1^0, \ldots, e_k^0$ of the dynamic constants and for all $t \geq 1$,

$$\begin{array}{llll}
\theta \ulcorner e_1^0 \urcorner \ldots \ulcorner e_k^0 \urcorner & \rightarrow_{Kt} & \theta \ulcorner e_1^t \urcorner \ldots \ulcorner e_k^t \urcorner & \left\{ \begin{array}{l} \textit{if the run does not halt} \\ \textit{nor fail nor clash} \\ \textit{for steps} \leq t \end{array} \right. \\
\theta \ulcorner e_1^0 \urcorner \ldots \ulcorner e_k^0 \urcorner & \rightarrow_{Ks} & \langle \ulcorner 1 \urcorner, \ulcorner e_1^s \urcorner \ldots \ulcorner e_\ell^s \urcorner \rangle & \textit{if the run halts at step } s \leq t \\
\theta \ulcorner e_1^0 \urcorner \ldots \ulcorner e_k^0 \urcorner & \rightarrow_{Ks} & \ulcorner 2 \urcorner & \textit{if the run fails at step } s \leq t \\
\theta \ulcorner e_1^0 \urcorner \ldots \ulcorner e_k^0 \urcorner & \rightarrow_{Ks} & \ulcorner 3 \urcorner & \textit{if the run clashes at step } s \leq t
\end{array}$$

Thus, groups of K successive reductions simulate in a simple way the successive states of the ASM, and give the output in due time when it is defined.

Proof. Use Theorem 2.18 to normalize the program P. We stick to the notations of that Theorem. Since there is no dynamic function, only dynamic constants, the ASM terms C_i and $I_{i,j}$ name the result of applying to the dynamic constants a composition of the static functions (including static constants). Thus, one can associate good $\mathbb{F}$-terms $\rho_i, \varphi_{i,j}$ to these compositions.
Observe that one can decide if the program halts or fails or clashes via some composition of functions in $\mathbb{F}$ (use the static equality function which has been assumed, cf. Definition 2.14). So enter negative answers to these decisions in the existing conditions $C_1, \ldots, C_n$. Also, add three more conditions to deal with the positive answers to these decisions. These three last conditions are associated to terms $\gamma_1, \gamma_2, \gamma_3$. Finally, apply Lemma 4.5 (with $p = n$ and $q = 3$). $\square$

Remark 5.3. A simple count in the proof of Lemma 4.5 allows to bound K_0 as follows: $K_0 = O((\text{size of } P)^2)$.

5.4 Main Theorem for All ASMs

Let ψ be a dynamic symbol. Its initial interpretation $\psi_{\mathcal{S}_0}$ is given by a composition of the static objects (cf. Definition 2.5) hence it is available in each successor state of the initial state. In subsequent states $\mathcal{S}_t$, its interpretation $\psi_{\mathcal{S}_t}$ is different but remains almost equal to $\psi_{\mathcal{S}_0}$: the two differ only on finitely many tuples. This is so because, at each step, any dynamic symbol is modified on at most N tuples where N depends on the program. Let $\Delta\psi$ be a list of all tuples on which $\psi_{\mathcal{S}_0}$ has been modified. What can be done with ψ can also be done with $\psi_{\mathcal{S}_0}$ and $\Delta\psi$. Since $\psi_{\mathcal{S}_0}$ is available in each successor state of the initial state, we are going to encode $\Delta\psi_{\mathcal{S}_t}$ rather than $\psi_{\mathcal{S}_t}$. Now, $\Delta\psi_{\mathcal{S}_t}$ is a list and we need to access in constant time any element of the list. And we also need to manage the growth of the list.
This is not possible in constant time with the usual encodings of datatypes in Lambda calculus. So the solution is to make $\Lambda_{\mathbb{F}}$ bigger: put new constant symbols to represent lists and allow new $\mathbb{F}$-reduction axioms to get *in one step* the needed information on lists.
Now, is this fair? We think it is as regards simulation of ASMs. In ASM theory, one application of the program is done in one unit of time though it involves a lot of things to do. In particular, one can get in one unit of time all needed information about the values of static or dynamic functions on

the tuples named by the ASM program. What we propose to do with the increase of $\Lambda_{\mathbb{F}}$ is just to get more power, as ASMs do on their side.

Definition 5.4. *Let $A_1, \ldots, A_n$ be the datatypes involved in functions of $\mathbb{F}$. If $\varepsilon = (i_1, \ldots, i_m, i)$ is an $(m+1)$-tuple of elements in $\{1, \ldots, n\}$, we let L_ε be the datatype of finite sequences of $(m+1)$-tuples in $A_{i_1} \times \cdots \times A_{i_m} \times A_i$. Let E be a family of tuples of elements of $\{1, \ldots, n\}$. The Lambda calculus $\Lambda_{\mathbb{F}}^E$ is obtained by adding to $\Lambda_{\mathbb{F}}$ families of symbols*

$$(F_\varepsilon, B_\varepsilon, V_\varepsilon, Add_\varepsilon, Del_\varepsilon)_{\varepsilon \in E}$$

and the axioms associated to the following intuitions. For $\varepsilon = (i_1, \ldots, i_m, i)$,

i. *Symbol F_ε is to represent the function $L_\varepsilon \to$ `Bool` such that, for $\sigma \in L_\varepsilon$, $F_\varepsilon(\sigma)$ is `True` if and only if σ is functional in its first m components. In other words, F_ε checks if any two distinct sequences in σ always differ on their first m components.*

ii. *Symbol B_ε is to represent the function $L_\varepsilon \times (A_{i_1} \times \cdots \times A_{i_m}) \to$ `Bool` such that, for $\sigma \in L_\varepsilon$ and $\vec{a} \in A_{i_1} \times \cdots \times A_{i_m}$, $B_\varepsilon(\sigma, \vec{a})$ is `True` if and only if $\vec{a}$ is a prefix of some $(m+1)$-tuple in the finite sequence σ.*

iii. *Symbol V_ε is to represent the function $L_\varepsilon \times (A_{i_1} \times \cdots \times A_{i_m}) \to A_i$ such that, for $\sigma \in L_\varepsilon$ and $\vec{a} \in A_{i_1} \times \cdots \times A_{i_m}$,*
- $V_\varepsilon(\sigma, \vec{a})$ is defined if and only if $F_\varepsilon(\sigma) =$ `True` and $B_\varepsilon(\sigma, \vec{a}) =$ `True`,
- when defined, $V_\varepsilon(\sigma, \vec{a})$ is the last component of the unique $(m+1)$-tuple in the finite sequence σ which extends the m-tuple $\vec{a}$.

iv. *Symbol Add_ε is to represent the function $L_\varepsilon \times (A_{i_1} \times \cdots \times A_{i_m} \times A_i) \to L_\varepsilon$ such that, for $\sigma \in L_\varepsilon$ and $\vec{a} \in A_{i_1} \times \cdots \times A_{i_m} \times A_i$, $Add_\varepsilon(\sigma, \vec{a})$ is obtained by adding the tuple $\vec{a}$ as last element in the finite sequence σ.*

v. *Symbol Del_ε is to represent the function $L_\varepsilon \times (A_{i_1} \times \cdots \times A_{i_m} \times A_i) \to L_\varepsilon$ such that, for $\sigma \in L_\varepsilon$ and $\vec{a} \in A_{i_1} \times \cdots \times A_{i_m} \times A_i$, $Del_\varepsilon(\sigma, \vec{a})$ is obtained by deleting all occurrences of the tuple $\vec{a}$ in the finite sequence σ.*

Now, we can extend Theorem 5.2.

Theorem 5.5. *Let $(\mathcal{L}, P, (\xi, \mathcal{J}))$ be an ASM with base sets $\mathcal{U}_1, \ldots, \mathcal{U}_n$. Let $A_1, \ldots, A_n$ be the datatypes $A_1^{(0)}, \ldots, A_n^{(0)}$ (cf. Proposition 5.1). Let $\mathbb{F}$ be the family of interpretations of all static symbols of the ASM restricted to*

the datatypes A_1,..., A_n. Let $\eta_1, \ldots, \eta_k$ be the dynamic symbols of the ASM. Suppose η_i has type $\mathcal{U}_{\tau(i,1)} \times \cdots \times \mathcal{U}_{\tau(i,p_i)} \rightarrow \mathcal{U}_{q_i}$ for $i = 1, \ldots, k$.
Set $E = \{(\tau(i,1), \ldots, \tau(i,p_i), q_i) \mid i = 1, \ldots, k\}$.
The conclusion of Theorem 5.2 is still valid in the Lambda calculus $\Lambda^E_{\mathbb{F}}$ with the following modification:

> *e^t_i is the list of $p_i + 1$-tuples describing the differences between the interpretations of $(\eta_i)_{\mathcal{S}_0}$ and $(\eta_i)_{\mathcal{S}_t}$.*

References

[1] Allan J. Atrubin. A One-Dimensional Real-Time Iterative Multiplier. *Trans. on Electronic Computers.* EC-14(3):394-399 June 1965.

[2] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.

[3] Henk P. Barendregt. *The Lambda calculus. Its syntax and semantics.* North-Holland, 1984.

[4] Henk Barendregt and Richard Statman. *Böhm's Theorem, Church's Delta, Numeral Systems, and Ershov Morphisms.* Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday. Lecture Notes in Computer Science 3838:40–54, 2005.

[5] T. Biedl, J.F. Buss, E.D. Demaine, M.L. Demaine, M. Hajiaghayi and T. Vinař. Palindrome recognition using a multidimensional tape. *Theoretical Computer Science*, 302(1-3):475–480, 2003.

[6] Egon Börger. The Origins and the Development of the ASM Method for High Level System Design and Analysis. *Journal of Universal Computer Science*, 8(1):2–74, 2002.

[7] Alonzo Church and John B. Rosser. Some properties of conversion. *Trans. Amer. Math. Soc.*, 39:472–482, 1937.

[8] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[9] Haskell Curry and R. Feys. *Combinatory logic, vol I.* North Holland, 1958.

[10] Nachum Dershowitz and Yuri Gurevich. A natural axiomatization of computability and proof of Church's Thesis. *Bulletin. of Symbolic Logic*, 14(3):299–350, 2008.

[11] Yuri Gurevich. Reconsidering Turing's Thesis: towards more realistic semantics of programs. *Technical Report CRL-TR-38-84, EEC Department, University of Michigan*, 1984.

[12] Yuri Gurevich. A new Thesis. *Abstracts, American Math. Soc.*, 1985.

[13] Yuri Gurevich. Evolving Algebras: An Introductory Tutorial. *Bulletin of the European Association for Theoretical Computer Science*, 43: 264–284, February 1991. Reprinted in *Current Trends in Theoretical Computer Science, 1993*, 266–29, World Scientific, 1993.

[14] Yuri Gurevich. *Evolving algebras 1993: Lipari guide.* Specification and Validation Methods. Oxford University Press. 9-36, 1995.

[15] Yuri Gurevich. *May 1997 Draft of the ASM Guide.* Tech Report CSE-TR-336-97, EECS Dept, University of Michigan, 1997.

[16] Yuri Gurevich. The Sequential ASM Thesis. *Bulletin of the European Association for Theoretical Computer Science*, 67: 93–124, February 1999. Reprinted in *Current Trends in Theoretical Computer Science, 2001*, 363–392, World Scientific, 2001.

[17] Yuri Gurevich. Sequential Abstract State Machines capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[18] F.C. Hennie. One-tape off-line Turing machine complexity. *Information and Computation*, 8:553–578, 1965.

[19] Chris Hankin. *Lambda calculi. A guide for computer scientists.* Graduate Texts in Computer, Oxford University Press, 1994.

[20] Andrei N. Kolmogorov. On the definition of algorithm. *Uspekhi Mat. Nauk*, 13(4):3–28, 1958. *Translations Amer. Math. Soc.*, 29:217–245, 1963.

[21] Donald Knuth. *The Art of Computer Programming (vol. 2.* 3rd edition, Addison-Wesley, 1998.

[22] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order and Symbolic Computation*, 20:199–207, 2007.

[23] Torben Æ. Mogensen. Efficient Self-Interpretation in Lambda Calculus. *J. of Functional Programming*, 2(3): 345-363, 1992.

[24] Wolfgang Paul. *Kolmogorov complexity and lower bounds.* Second Int. Conf. on Fundamentals of Computation Theory. L. Budach editor, Akademie Berlin. 325–334, 1979.

[25] Simona Ronchi Della Rocca and Luca Paolini. The Parametric Lambda-calculus. A Metamodel for Computation. Springer-Verlag 2004, XIII, 252 p.

[26] Richard Statman. *Church's Lambda Delta Calculus.* LPAR 2000. Lecture Notes in Computer Science 1955:293–307, 2000.