Mémoire scientifique pour l'obtention d'une

Habilitation à Diriger des Recherches

présentée à

l'Université Paris Est Créteil

par

Pierre Valarcher

Titre :

Autour du concept d'algorithme: intensionnalité, complétude algorithmique, programmation et modèles de calcul

suivi de

fonctions booléennes et cryptographie

présentée le lundi 18 octobre 2010 devant le jury composé de

Rapporteurs		
	Etienne Grandjean	(Pr. Université de Caen)
	Yuri Gurevich	(Pr. University Michigan, Microsoft Research)
	Jean-Yves Marion	(Pr. Université de Nancy)
	Yiannis Moschovakis	(Pr. University California Los Angeles)
Examinateurs :		
	Patrick Cegielski	(Pr. Université Paris Est Créteil, directeur)
	Loic Colson	(Pr. Université de Metz)
	Hubert Comon	(Pr. ENS Cachan)
	Gilles Dowek	(Pr. Ecole Polytechnique)
	Jean-Francis Michon	(Pr. Université de Rouen)
	Dominique Perrin	(Pr. Université Paris Est Marne-la-vallée)

Contents

Chapter 1

Remerciements

Je tiens à remercier très chaleureusement Yuri Gurevich et Yiannis Moschovakis qui m'ont fait l'honneur d'accepter de lire et de commenter mon travail. Les échanges de ces derniers mois ont été une source importante de questionnements et de plaisirs.

Etienne Grandjean et Jean-Yves Marion sont pour moi des enseignants-chercheurs exemplaires, ils m'ont toujours soutenu et encouragé dans cette thématique. Je les remercie d'avoir accepté de rapporter ce mémoire.

Je suis resté en poste à l'université de Rouen pendant 10 ans. J'y ai rencontré de nombreuses personnes et plus particulièrement Jean-Francis Michon qui m'a emmené faire une excursion dans le monde de la cryptographie et des fonctions booléennes; sa manière d'appréhender les événements est un modèle pour moi. J'essaie de m'en inspirer le plus souvent possible. C'est avec une grande joie que je le revoie pour cette soutenance.

Patrick Cegielski m'a accueilli à l'IUT de Fontainebleau en 2006, il a accepté d'être directeur de ce mémoire sans aucune hésitation. Je lui serais toujours reconnaissant pour son soutien sans faille dans mes activités de pédagogie et de recherche.

Loic Colson est à l'origine de ce travail, qu'il soit grandement remercié d'avoir accepté d'être membre dans ce jury.

Je me rappelle de Dominique Perrin, comme directeur du LITP lorsque que j'étais étudiant puis doctorant. Il m'a permis de faire mes premières armes dans le monde de la recherche, toujours ouvert, généreux et curieux.

Gilles Dowek fait parti de ces personnes qui partagent leur temps entre la recherche et l'envie de transmettre leurs connaissances à un maximum de personnes. Je l'envie pour ses capacités pédagogiques et la qualité de ces écrits. C'est un grand plaisir de l'avoir dans mon jury.

Hubert Comon n'a pas hésité à participer à ce jury qu'il en soit remercié.

Sont venues les lignes où je dois exprimer mes remerciements à celui qui m'à suivi tout au long de ces années. Serge Grigorieff accompagne mes recherches depuis maintenant 18 ans. Au travers de notre collaboration j'ai beaucoup appris et je continue à beaucoup apprendre. Il m'a transmis une part de son immense goût pour la recherche. Serge, je t'exprime ma profonde amitié et je te dis simplement un immense merci!

Enfin, et surtout, je remercie Léo, Marion et Nadège qui m'ont soutenu et supporté durant toutes ces années. Qu'ils sachent que je pense constamment à eux et qu'ils sont une source d'inspiration permanente pour moi.

Chapter 2

Introduction

My works have as main objective to contribute to the formal study of the concept of algorithm. I choose to approach this issue from three angles: the **structural complexity** of some programming languages, the **algorithmic completeness** of two languages relatively to a class of algorithms and the **evolving multi-algebras** as a representation of models of sequential computation. These three themes are the subject of the first three chapters.

The final chapter is a contribution to the study of boolean functions and the representation of some of them using binary decision diagrams (OBDD).

To connect the concept of algorithm with its implementation, the concept of simulation is necessary. This simulation is more or less fine depending on the angle of approach we choose: a strict step-by-step simulation (one step of computation in one language is simulated by one step in the other language) from a imperative language and a functional language in the context of structural complexity, a non-strict step-by-step simulation (a computational model is simulated by at most k steps in the other model) between algorithms and programming languages in the chapter on algorithmic completeness and finally a literal "identification" (all elements of the simulated model have part of the simulation model with just a change of viewpoint) between "all"[1] models of computation and a model based on abstract state machines, Y. Gurevich ([Gur93]).

The starting point of the work has its origins in the study of the **algorithmic deficiencies** of some programming languages (the limits of some programming languages to simulate some algorithms). The first problem that has been proved on a gap between algorithms and program is called the algorithmic problem of *min* (it is due to Loïc Colson in the late 1980s [Col89]) it may be introduced like this

> being given a set of functions (in his case, the class of primitive recursive functions PR) and a programming language to compute them all (in his case, the language of combinators Pr), does there exist a program that implements the following algorithm[2] (which calculates the minimum of two integers):
>
> > *each integer is represented in unary on a tape of a Turing machine (two tapes are necessary) and program of the Turing machine proceeds as follows: each cell is read alternatively (in traveling from left to right for example)*

[1] although this assertion can not be proved but only to be supported by a large number of instances. Exactly like the idea of Church thesis.

[2] It may be noted that the algorithm is expressed in a formalism (here in terms of a Turing machine) different from the programming language being studied (which may, in turn, be presented as a system of recursive equations).

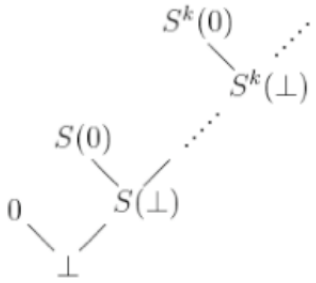

Figure 2.1: The lazy natural integer domain.

> *and when the value read (on one tape) is different from the value on the other tape, so the state of the machine becomes final (it is indexed, for example, by the number of the tape and then indicates the smallest integers). Initially heads are placed leftmost of the integers.*

The answer given by L. Colson, to this problem is negative and involves a property (known as the *ultimate obstinacy* property) which is formalized using denotational semantics in a particular domain (in this case the lazy integer domain Fig. 2.1) and which establishes that the language (PR) cannot simulate alternation (ultimately) of the "path" of its inputs. The corollary is a property of computational complexity, which proves that there can't exist a program, in the language PR, that calculates the minimum of two integers in linear time in the minimum of the size of the inputs.

This result is the first to establish that some programming languages have algorithms weaknesses to compute some functions (so that those functions are computable in this language by other algorithms). In other words, some languages do not allow to simulate some algorithms.

Further languages are available (for the same class of functions) some cure this shortcoming, some check other similar properties.

All these studies on algorithmic flaws of programming languages continue throughout the decade of 90 with Loïc Colson [Col96],[CF98] Daniel Fredholm [Fre96], René David [Dav93, Dav94, Dav97, Dav03], Yannis Moschovakis [Mos03], Lou Van Dries [Dri03]; classes of functions studied are the class of primitive recursive functions ([Pét68]) on the one hand and the class of functions provably total in Peano arithmetic (primitive recursive functional [GLT89]) on the other hand. These classes are also those on which the work presented here is based.

In [Mos03], Y. Moschovakis suggests to call this study **structural complexity of algorithms** and raises the fundamental question of the **formulation of algorithms** which remains vague for L. Colson and R. David and which is based on definitions of recursive equations in the Herbrand-Gödel style to D. Fredholm and Y. Moschovakis.

Structural complexity of programming languages

The fact that all these deficiencies have been established from the study of functional languages could have inferred that this occurred only in the functional world. However, similar results are established for some imperative languages. This is done in the first chapter ([CLV06]) of this thesis for the language LOOP of Meyer and Ritchie ([MR76]) and for an imperative language LOOP$^\omega$

(that is, to our knowledge, the only mandatory language that captures the system T, [CPV09]) to represent exactly the functions of the system T of Gödel (primitive recursive functionals). It is interesting to note that these results on imperative programming languages are obtained by the translation of imperative programs into functional programs that simulate step-by-step (strictly), thereby "recover" the properties and shortcomings of these languages and to extend them to imperative languages.

The fact that these deficiencies are extended to imperative languages shows that it is important to study the strengths and weaknesses of control structures in programming languages (schema of recursion, iterations bounded, the presence of conditional, ...).

The fact remains that the question of the definition of a class of algorithms arises in order to achieve more "positive" in this issue: namely, given a class of algorithms, does there exist a programming language that simulates step-by-step all these algorithms.

Algorithmic completeness

In chapter two, I propose the definition of a class of algorithms for primitive recursive functions (PRA) and then I define a imperative programming language and a functional programming language that capture all the algorithms of this class of algorithms. The simulation is not strict because algorithms perform operations simultaneously (in finite bounded), operations that are not allowed in usual programming language (even theoretical one).

To define the class of algorithms I relied on the formal definition of algorithm given by Y. Gurevich ([Gur85, Gur93, Rei03b]), namely the abstract state machines (ASM) which define an algorithm as a transition system whose states are first-order logical structures. These transitions (definable using a program built from three kinds of rules) modify at each step, a bounded finite number of elements of the state, need to know a bounded finite number of values of the state and does not change the signature. This model can capture all the algorithms even partial ([Gur00, Rei03a])[3]

To limit the class of PRA algorithms to compute primitive recursive functions only, we chose to restrict the number of transitions to make it computable by a primitive recursive function and restrict the vocabulary from which those algorithms are built (the restriction we choose for the operations on which are built algorithms is not sufficient to limit the computation to primitive recursive functions).

The two definitions of algorithms (that of Y. Gurevich and that of Y. Moschovakis) agree on the fact that an algorithm is not absolute but is relative to the level of abstraction chosen (a set of functions over some algebras). Here, we limit ourselves to a ultra-simple arithmetic (successor, predecessor, equality).

The PRA class is then the set of couple consisting of an ASM program and a primitive recursive function (representing a milestone in computing time, here, the number of transitions to the computation. The ASM program alone may compute a non primitive recursive function or a non total function). Languages that allow the simulation of the algorithms of this class is an extension of the language Loop with an instruction to "escape" from bounded iterations in the imperative framework ([APV]) and a restricted version of the system T (we can use a

[3]There are only two formal definitions of the notion of algorithms, one comes from the theory of machines (Y. Gurevich with Abstract State Machine), the second definition given by Y. Moschovakis ([Mos98, Mos01, MP08]) comes from theory of recursive equations. Y. Moschovakis identifies an algorithm as a system of recursive equations and developed a theory of recursor. This approach is more extensional, at first, but with a focus on the time complexity of algorithms, than the one of Y. Gurevich. You can see comparisons in the critical work of W. Dean ([Dea07], criticism of the two definitions). The choice I made is motivated by the study of imperative languages. The ASM model is "closest" to imperative languages than functional languages. However, in [MV09], we establish a first connection between the class of PRA algorithms and system of recursive equations.

single level of functionality) with a call-by-value strategy in the case of the classical reduction (called β–reduction) and a call-by-name strategy in the case of reducing the recursion in the functional framework ([MV09]). The functional language may in fact be represented by a system of equations similar to the recursion with variable parameters ([Pét68]).

These results establish a notion of **algorithmic completeness** of some programming languages that also are extensionally complete (they compute exactly the primitive recursive functions).

Evolving multialgebra

The third chapter takes up the theme of the simulation but the topic is not on programming languages, but the computational models and their representation within the ASM framework. This work comes from a dissatisfaction with the way the usual models of computation (Turing machine, RAM, Grammar, ...) were sometimes shown simulated (not strictly-by-step, see [BS03] example) by ASM.

Indeed, the usual choice of representation can be accomplished by placing in the state of the algorithm to be simulated the program and the program (of the ASM) is therefore a universal model simulator. This representation can be "improved" by letting in the state the representations of model elements (and only them) and to express only valid programs (syntactically) in place of the ASM program. This allows for a real identification of the model objects with mathematical objects (expressible with first order).

This approach, first tested with the model of Turing machines, has to be powerful enough to represent all the usual sequential models. It leads to the transformation of ASM to evolving multialgebras (recalling the first name given to ASM, "Evolving algebra"). The symbols of the state are typed (to restrict the composition of functions) and the program is an abstract functional similar to a function of transitions of a Turing machine (symbols are updated according values in a finite number of points or other such symbols).

This approach presented in [GV10] allows a unified representation of different sequential models of computation and evolving multialgebras can be regarded as the model unifying models of sequential computations.

Boolean functions

The last chapter is independent of the previous chapters. It is the summary of research done at the University of Rouen during the period 2003-2006.

This is a study of boolean functions and their applications to cryptography. This work is part of the attempted at cryptanalysis ([MVY03]) of public key encryption HFE (Hidden Field Equation, [14]). This attempt at cryptanalysis was conducted in trying to solve the system of boolean equations using binary decision diagrams (BDD). Despite the failure of this attempt, we analyzed the representation of a particular class of boolean functions using OBDD and this leads to a characterization of boolean functions (called "hard') and BDD representatives ([MVY05]).

Nota Bene. This manuscript is a summary of my main research activities. Some articles on sub-themes ([Val05], [GMV10]) have been deliberately left out, such as those made during my Phd thesis ([Val00],[Val08]).

Chapter 3

Structural complexity of programming languages

3.1 Introduction

In this chapter, we introduce a statically typed extension of the LOOP language (called LOOP$^\omega$) with higher-order procedures and procedural variables and we argue that this programming language is a natural imperative counterpart of Gödel System T. The argument is two-fold:

1. we define a translation of the LOOP$^\omega$ language into System T and we prove that this translation actually provides a step by step simulation,
2. using a converse translation, we show that LOOP$^\omega$ is expressive enough to encode any term of System T.

The translation is faithful and allows us to give two outcomes:

- a first application of these results is extensional: we derive a new characterization of the class of Csillag-Kalmar elementary functions (the class $\mathcal{E}_3$ in Grzegorczyk hierarchy [Grz53]). In [BW00], such a characterization is based on a syntactic restriction on terms of a variant of Gödel System T. As a corollary of various properties of our two translations we provide an imperative counterpart of this restriction. In particular, we obtain that any LOOP$^\omega$ program in which any bound of loop is a read-only input variable is elementary.
- a second application is intensional and is related to the so-called minimum problem. In [CF98], L. Colson and D. Fredholm proved that in call-by-value System T (arguments are evaluated first), any algorithm which computes a non-trivial binary function (where trivial means constant or projection plus constant), has a time-complexity which is at least linear in one of the inputs. As a consequence of this property, there is no term that computes the minimum of two natural numbers n and m in time $O(min(n, m))$. As a corollary of the step by step simulation we obtain a similar negative result for LOOP$^\omega$ programs.

3.2 System T

Primitive recursive functionals of finite types, also known as Gödel system T in logic (see [GLT89]), and as typed lambda calculus with primitive recursive recursion in higher types in computer science, are quite important in both areas. In the former they are used in proof-theoretical

studies of Peano arithmetic, in the latter they can be used to give the formal semantics of some modern kind (functional) computer programming languages. We give the definition of system T with product types (tuples and n-ary functions), a constant-time predecessor operation:

Definition 1. *We define* types *and* typed-terms *by induction:*

Types *are defined inductively by: N is the type of natural numbers, and if U and V are types then also are $U \to V$ and $U \times V$;*

Typed–terms *$t : T$ are defined inductively*

- *for each types T, the set of typed variables $x^T : T, \mathtt{y}^T : T, \ldots$ are typed–terms,*
- *$0 : N$ is a typed–term,*
- *$S(t) : N$ and $\mathtt{pred}(t) : N$ are typed–terms if $t : N$ is a typed–term,*
- *$(t_1, \cdots, t_n) : (T_1 \times \cdots \times T_n)$ is a typed–term if $t_1 : T_1, \cdots, t_n : T_n$ are typed–terms,*
- *$\lambda(x_1^{U_1}, \cdots, x_n^{U_n}).t : (U_1 \times \cdots \times U_n) \to T$ is a typed–term if $t : T$ is a typed–term,*
- *$t\ u : T$ is a typed–term if $t : U \to T$ et $u : U$ are typed–terms,*
- *$\mathtt{rec}_T(t,\ b,\ [x^N, y^T]\ s) : T$ is a typed–term if $t : N$, $b : T$, $s : T$ are typed–terms.*

We choose a special syntax for the *rec* that has been, first, introduced by L. Colson in [Col91, Col96]. Beyond the theoretical (philosophical) point of view[1] this presentation allows us to consider, easily, two different strategies of reductions : one for the β–reduction and one for the *rec*[2].

The reduction strategy for our purpose is the call-by-value strategy (the evaluation is arguments first from left to right). Then $[x, y]\ s$ in `rec` is an other notation for $\lambda x.\lambda y.s$.

3.3 Loop$^\omega$

The Loop language [MR76] is a core imperative language in which programs consist only of assignments, sequences, and bounded loops. Meyer and Ritchie proved in particular that Loop programs compute exactly the class of primitive recursive functions. The Loop language has since been widely studied in the literature (see for instance the textbook [DW83]).

The Loop language is extended with first-class procedures (true closures) and mutable procedural variables (aka function pointers). This language is called Loop$^\omega$. It is a *pure* imperative language in the following sense: its type system forbids side-effects and parameter-induced aliasing. For a better readability, we omit types in the following definition (where $\bar{q}$ denotes natural integer, blocks follows the C language syntax and we consider only two formal parameter modes `in` and `out`):

[1] As (private communication) noticed by L. Colson in [Col91], in the usual presentation of system T, to deduce that $\Gamma \vdash rec_T(n, b, s) : N$ we need the three hypothesis $\Gamma \vdash n : N$, $\Gamma \vdash b : T$ and $\Gamma \vdash s : N \to T \to T$ then we need the $\to$ to define the rule for recursion. In our case, the rule is $\Gamma, x : N, y : T \vdash s : T$ no arrow appears in the rule.

[2] This will be used in Ch.4.

Definition 2 (Syntax of $Loop^{\omega}$)**.**

$$
\begin{array}{rrcl}
(\textit{command}) & c & ::= & \{s\} \\
 & & | & \texttt{for}\ y\ := 1\ \texttt{to}\ e\ \{s\} \\
 & & | & y := e \mid \texttt{inc}(y) \mid \texttt{dec}(y) \\
 & & | & p(\vec{e};\vec{y}) \\
(\textit{sequence}) & s & ::= & \epsilon \\
 & & | & c;\ s \\
 & & | & \texttt{cst}\ y\ =\ e;\ s \\
 & & | & \texttt{var}\ y\ :=\ e;\ s \\
 & & & \\
(\textit{anonymousprocedure}) & a & ::= & \texttt{proc}(\texttt{in}\ \vec{y};\texttt{out}\ \vec{z})\ \{s\} \\
 & & & \\
(\textit{expression}) & e & ::= & y \mid \bar{q} \mid a \\
(\textit{procedure}) & p & ::= & y \mid a \\
(\textit{value}) & w & ::= & \bar{q} \mid a
\end{array}
$$

Constraints imposed by the type system [CPV09] allows us to give a simple operational semantics in terms of transition system which we prove to be equivalent to the natural semantics.

An other operational semantics is given through the translation of a program of Loop$^{\omega}$ onto a term of System T. This translation preserves the operational semantics and therefor we obtain a step-by-step simulation.

3.4 Translation

For sake of simplicity we write $\texttt{let}\ (x_1,\ldots,x_n) = u\ \texttt{in}\ t$ be an abbreviation for the redex $\lambda(x_1,\ldots,x_n).t\ \ u$. Then the intuition behind the translation is that, if $\vec{x}$ denotes variables of the environment then the block $\{c_1;\ldots;c_n\}$ is translated into

$$\texttt{let}\ \vec{x} = c_1^*\ \texttt{in}\ldots\ \texttt{let}\ \vec{x} = c_n^*\ \texttt{in}\ \vec{x}$$

Where the * operation is defined below.

We may be more precise as in [CPV09] and annotate blocks by the set of variables that have their value modified, for intance the block $\{\texttt{inc}(x);\texttt{inc}(x)\}_x$ is translated as

$$\texttt{let}\ x = \texttt{succ}(x)\ \texttt{in}\ \texttt{let}\ x = \texttt{succ}(x)\ \texttt{in}\ x$$

The translation is defined by

Definition 3. *The* translation e^* *and* $\{s\}^*_{\vec{x}}$ *of respectively an expression* e *and a block* $\{s\}_{\vec{x}}$ *into a term of System T are defined by mutual induction:*

- $\bar{n}^* = S^n(0)$
- $y^* = y$
- $\{\}^*_{\vec{x}} = \vec{x}$
- $(\texttt{proc}(\texttt{in}\ \vec{y};\texttt{out}\ \vec{z})\{s\}_{\vec{z}})^* = \lambda\vec{y}.\{s\}^*_{\vec{z}}[\vec{z}_0/\vec{z}]$ *where* $\vec{z}_0$ *denotes the default value for each type of* z *variables.*
- $\{\texttt{var}\ y := e;\ s\}^*_{\vec{x}} = \{s\}^*_{\vec{x}}[e^*/y]$

- $\{\texttt{cst } y = e;\ s\}^*_{\vec{x}} = \texttt{let } y\ =\ e^* \texttt{ in } \{s\}^*_{\vec{x}}$
- $\{y := e;\ s\}^*_{\vec{x}} = \texttt{let } y\ =\ e^* \texttt{ in } \{s\}^*_{\vec{x}}$
- $\{\texttt{inc}(y);\ s\}^*_{\vec{x}} = \texttt{let } y\ =\ \texttt{succ}(y) \texttt{ in } \{s\}^*_{\vec{x}}$
- $\{\texttt{dec}(y);\ s\}^*_{\vec{x}} = \texttt{let } y\ =\ \texttt{pred}(y) \texttt{ in } \{s\}^*_{\vec{x}}$
- $\{p(\vec{e};\vec{z});\ s\}^*_{\vec{x}} = \texttt{let } \vec{z}\ =\ p^*\ \vec{e}^* \texttt{ in } \{s\}^*_{\vec{x}}$
- $\{\{s_1\}_{\vec{z}};\ s_2\}^*_{\vec{x}} = \texttt{let } \vec{z}\ =\ \{s_1\}^*_{\vec{z}} \texttt{ in } \{s_2\}^*_{\vec{x}}$
- $\{\texttt{for } y :=\ 1 \texttt{ to } e\ \{s_1\}_{\vec{z}};\ s_2\}^*_{\vec{x}} = \texttt{let } \vec{z}\ =\ \texttt{rec}(e^*, \vec{z}, [\vec{z}, y]\{s_1\}^*_{\vec{z}}) \texttt{ in } \{s_2\}^*_{\vec{x}}$

We obtain then the following theorem:

Theorem 1. *For any well-typed state (c, μ) (μ is the environment), if $\vec{x} = dom(\mu)$ we have:*

$$(c, \mu) \to (c', \mu') \quad \textit{implies} \quad c^*[\mu(\vec{x})^*] \rightsquigarrow c'^*[\mu'(\vec{x})^*]$$

Recall that a major result concerning System T states that functions on the natural numbers that are definable in this system correspond exactly to functions that are provably total in first-order Peano arithmetic (see the textbook [Sch67]). More precisely, that there is a syntactic hierarchy of fragments T_n of System T such that the class of functions representable in T_n is identical to the class of functions provably recursive in the fragment of Peano arithmetic where induction is restricted to Σ_{n+1} sentences. In particular, T_0 corresponds to the class of primitive recursive functions. We define thus a similar hierarchy of fragments $\textsc{Loop}_n$ of $\textsc{Loop}^\omega$ and we show that both translations relate programs of $\textsc{Loop}_n$ and terms of T_n. As a corollary, we obtain that the functions representable in a language with higher-order procedures but without procedural variables (which is a sub-language of $\textsc{Loop}_0$) are primitive recursive. This corollary generalizes thus previous results presented in [CLV06] where Meyer and Ritchies Loop language was translated into T_0. On the other hand, the Ackermann function which is known not to be primitive recursive is representable in T_1 and thus also in $\textsc{Loop}_1$. As far as we know, $\textsc{Loop}^\omega$ is the first total imperative language allowing to program the Ackermann function.

3.5 Conclusion

This is the first time, as far as we know, up to now, that an imperative language is presented for the system T of Gödel. The translations used in this work take benefit of all the research in operational and denotational semantics functional programming. This work has been continued by T. Crolard and al. ([CP09]) by adding non local-jump and dependent types.

The fact that intensional behavior results can also be obtained (using a step-by-step preserving translation) improved the thesis that imperative programming languages can be also study from an algorithmic point of view and not only from a extensional point of view.

Chapter 4

Algorithmic completeness

4.1 Context

In [Col89], L. Colson using non-standard denotational semantics (the used domain is the *lazy natural integers* [Esc93]) proves that, though the function *min* which computes the minimum of two integers is obviously a primitive recursive function (see [Pét68] for a formal definition), there is no way to represent, in the model of primitive recursive programs (which are called PR-combinators), the better known algorithm, which decreases alternatively both arguments. The main reason is a so called ultimate obstinacy theorem showing that every PR-combinator must choose one (and only one) of its arguments and thus the alternation between arguments is impossible. A constructive proof of this property can be found in [Coq92]. This work, despite the fact that complexity results may be obtained, has opened some researches on the behavior of programs and especially functional programs. Following L. Colson, R. David (see [Dav01]) developed a new semantics (the trace of computation) allowing him to prove a new property (the backtracking property) for any primitive recursive program using any kind of data types. This trace semantics has been applied to usual schemas of primitive recursion and has given some results on set of functions that are behaviors and has allowed to compare the behavior of some primitive recursive schemas (see [DV10]).

Some new results have been proved on *intensional behavior* (also called *structural complexity*) about other primitive recursive schemas (in [Val96], [Val00] and [Val08]).

In the same framework, L. Colson and D. Fredholm (see [Fre96] and [CF98]) have shown that call-by-value strategy (with primitive recursion over lists of integers and with primitive recursion in higher types, for instance system T of Gödel) does not allow to implement the better known algorithm of the *min* function.

Similar questions have been studied by S. Brookes and D. Dancanet in [BD95] and [DB96] with non-determinism and CDS languages.

In [Mos03], Y. Moschovakis extended the Colson's result and proved that, even with the addition of a set of programs with *corrects* complexity (such as $<, ..$), any *logtime* algorithm cannot be implemented for the greatest common divisor (such as Stein's one, [CLRS01]).

Y. Moschovakis ended by an open problem relative to the classical Euclidean algorithm (L. Van Den Dries gives a partial answer in [Dri03]).

In [CLV06], the *min* problem is studied in an imperative framework: the Loop language which computes only primitive recursive functions ([MR76]). But there is no good program for the *min* function too. This result has been extended in the framework of language with higher-order procedural variables ([CPV09]). See section 3.2 for a description of the language and the

techniques used.

Then the limitation (from an algorithmic expressivity point of view) of a large kind of primitive recursive languages has been proved also, even imperative or functional programming language, do not allow to implement very simple algorithms, for instance simple algorithm for inf and gcd.

Our motivation is then to construct a more expressive language and prove that it captures integrally a class of algorithms that specify primitive recursive functions.

4.1.1 Results

We construct a set of algorithms from the notion of *Abstract State Machines* ([Gur93]). To compute only primitive recursive algorithms, we restrict the framework of the algorithms to usual primitive recursive vocabulary (unary integers with successor and predecessor, essentially). The number of steps of computation is also bounded to primitive recursive functions.

Then we construct two programming languages ([APV] and [MV09]) that are faithful to represent those algorithms. The two languages come from the imperative and functional paradigm.

4.1.2 A class of algorithm for primitive recursion

So far the notion of algorithms was defined as *an effective procedure that solves a problem* (this is the "definition" that occurs in most of books on algorithms).

This concept is intuitive and it is mostly reduced to the existence of a *mechanism* for realizing it (again we mean by mechanism, a mechanical process, such as a Turing machine or an automaton, or still possible to express the algorithm with a program that itself can be compiled into a mechanical process). For a philosophical and/or mathematical discussion of algorithm we can see [Dea07].

Nowadays only two attempts of the definition of algorithms arise. They are both influenced by classical programming : imperative and functional. They share the fundamental point that all algorithms depend of a level of abstraction that is designed by terms algebras.

Moschovakis algorithms The approach due to Y. Moschovakis ([Mos98],[Mos01] and [MP08]), considers that algorithms are expressed via definitions by mutual recursion and the algorithmic content of a such definition A is given by the semantics of equations (given by the recursor of a special form of A). For insance, with the algebra containing $(-, <)$, the following equation represents an algorithm for the gcd function:

$$gcd(x, y) = if(x = 0, y, if(y = 0, x, if(x < y, gcd(x, y - x), gcd(x - y, y))))$$

Abstract State Machine This definition was proposed by Y. Gurevich in [Gur85], [Gur93], [Gur00] and [DG08]; an axiomatic definition is mapped to the notion of *abstract state machine* with a *strict step-by-step* simulation (see [DDG97] for a definition of simulation and strict lockstep. The Abstract State Machines are a kind of super Turing machine that work not on simple tape (with finitely alphabets) but on multi–sorted algebras (see the point of view in [GV10]). A program is a finite set of rules that updates terms. For instance, if we consider an algebra with symbols $(res, x, y, <, -)$ then the following program is an algorithm for a gcd function:

`if` $x = 0$ `then` $res := y$
`if` $\neg(x = 0) \wedge y = 0$ `then` $res := x$
`if` $\neg(x = 0) \wedge \neg(y = 0) \ \wedge \ (x < y)$ `then` $y := y - x$
`if` $\neg(x = 0) \wedge \neg(y = 0) \ \wedge \ \neg(x < y)$ `then` $x := x - y$

4.2 Abstract State Machine (ASM)

The theory of ASM began in 1985 in [Gur85] but it was preceded by attempts due to Gandy ([Gan80]) and Schönhage ([Sch80]) among others. The ASMs are presented and debated in [Gur00],[BS03], [Rei03b] for instance. The main goal is not to define a new theoretical model for a computation model but a theoretical model for the notion of algorithm.

Definition 4. *An* Abstract State Machine *(in normal form) is defined by:*

- *a family of states S, which share the same vocabulary V,*
- *a set $I \subseteq S$ of initial states*
- *a program π, constructed by a finite sets (*simultaneously fired*) of command (*conditional *and* update*):*

$$\texttt{if}\ \ p\ \texttt{then}\quad f(t_1,\ldots,t_n) := t_0$$

with $f \in V$ and closed-terms t_i on V and p a conjunction equality or inequality between closed terms.

The program π allows to go from state S_i to state S_{i+1} with the following semantics: if $[\![p]\!]_{S_i} = true$ and if $[\![\vec{t}]\!]_{S_i} = \vec{a}$ then $[\![f(\vec{a})]\!]_{S_{i+1}} = [\![t_0]\!]_{S_i}$. Else $[\![f(\vec{a})]\!]_{S_{i+1}} = [\![f(\vec{a})]\!]_{S_i}$.

$f(t_1,\ldots,t_n) := t_0$ is called an *update* statement. In an ASMs program, more than one guard (conditions) may be true simultaneously. Then the updates are done simultaneously (recall that there is always a bounded simultaneous update). For instance the following program with $V = \{x, y\}$

$$\texttt{if}\ true\ \texttt{then}\ x := y$$
$$\texttt{if}\ true\ \texttt{then}\ y := x$$

allows to swap the values of two variables (without auxiliary variable nor computation).

Remark 1.

- *One may be more liberal by allowing simultaneous updates or a full conditional (all this is syntactic sugar). We use $||$ for the simultaneous updates and the usual* `if` p `then` P `else` Q *(see the paragraph 1 for an example).*
- *The vocabulary may be divided in at least two sorts of symbols : static and dynamic*[1]*. Dynamic symbols are those that will evolved during the computation. Static symbols do not evolved.*
- *When stops the computation? We consider at least two conditions*[2] *that halt a computation: the first one is when there is no evolution of the state after a run ($\pi(S_i) = S_i$), an other one is when we try to modify (simultaneously) the value of a dynamic symbol with two different values ($x := 3 \ || \ x := 4$ for instance). The previous stop is a kind of* clash *in the computation.*

Sample 1. *To illustrate a concrete algorithm, we anticipate a notion of ASM to be more structured (see Ch.5):*

- *one adds a dynamic variable (state) representing the status of the run: state $\in \{run, stop\} = K$ for instance;*

[1] In fact, one can consider only a sole set of symbols. The program allows to determine those that are updated (dynamics) from those that are not (statics).

[2] To be closed to programming languages

- *one considers multi-algebras (by typing functions for instance) and then allowing multi-copy of same set:* $< \mathbb{N}_1, succ >, < \mathbb{N}_2, pred >$*;*
- *one separates in three signatures the vocabulary of the algebra:* S^{int}_{sta} *for the internal static vocabulary,* S^{ext}_{sta} *for the external static vocabulary and* S^{int}_{dyn} *for the internal dynamic vocabulary (for instance* $state \in S^{int}_{dyn}$*).*

An algorithm is then given by $(S^{int}_{sta}, S^{ext}_{sta}, S^{int}_{dyn}; D; M; \pi)$ *with* D *a set of types (with* $K \in D$ *and* $\mathbb{B}$ *the boolean type),* M *the model (the interpretation of symbols) and* π *the program:*

`if` $x = 0$ `then` $res := y \parallel state := stop$
`else if` $y = 0$ `then` $res := x \parallel state := stop$
`else if` $y < x$ `then` $x := x - y$
`else` $y := y - x$

$D = \{\mathbb{N}, \mathbb{B}, K = \{run, stop\}\}$
$S^{int}_{sta} = \{(= 0) : \mathbb{N} \Rightarrow \mathbb{B}; (-) : \mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}; (\wedge) : \mathbb{B} \times \mathbb{B} \Rightarrow \mathbb{B}; \neg : \mathbb{B} \Rightarrow \mathbb{B}\}$
$S^{ext}_{sta} = \{N : \mathbb{N}, M : \mathbb{N}\}$
$S^{int}_{dyn} = \{x : \mathbb{N}, y : \mathbb{N}, res : \mathbb{N}, state : K\}$
$M = \{$*static symbols are usually interpreted*;
dynamic symbols have the following initialization : $x = N, y = M, res = 0, state = run\}$

ASMs have the characteristic to simulate all algorithms (due to the strength of state structure). This structure of state represents the degree of abstraction of the algorithms.

In [Gur00] and [DG08], Y. Gurevich gives three postulates for the definition of algorithms that can be summed up by

- an algorithm is a transition system
- states of an algorithm are first order structures
- each transition does a bounded work

From those postulates, Y. Gurevich shows that

Theorem 2 (Gurevich [Gur00])**.** *All algorithms (object that respect the three previous postulates) can be simulated* step by step *by an ASM.*

This result is then a starting point to consider a smaller class of algorithms that I am interested in: the class Apra of primitive recursive algorithms. Then we formulate the problem of algorithm expressivity of programming language by

> Does there exist a programming language (functional or imperative) that simulates all the algorithms in Apra?

We then explore this question.

4.3 Arithmetical Primitive Recursive ASMs

The aim of this section is to give a formal definition for a class of algorithms we call *Arithmetical Primitive Recursive Algorithms* (Apra). The following presentation is similar to the notion of arithmetical ASM ([DG08]). According to Gurevich's thesis (ASM equals algorithm, [Gur00]) it is sufficient to define the concept of *Arithmetical Primitive Recursive* ASM.

We need to restrict the vocabulary. As a minor constraint, only we consider functions taking their values over $\mathbb{N}$. Hence we will restrict us to ASMs working over non-negative integers. Then we define:

Definition 5. $\mathbb{N}$-***typed*** *ASMs are ASMs whose non-logic part of the vocabulary only contains functions taking their values on* $\mathbb{N}$.

Definition 6. *An* $\mathbb{N}$*-typed* $\mathcal{F}$*-ASM* A *is called* ***basic*** *or* ***arithmetical*** *if the non-logic part of its vocabulary is reduced to:*

$$\{n_1, \ldots, n_j, pred, succ, x_1, \ldots, x_k\}$$

where:

- $n_1, \ldots, n_j$*,* $pred$ *and* $succ$ *are static and represent respectively natural constant operations which belong to* $\mathbb{N}$*, the unary predecessor and successor functions on* $\mathbb{N}$ *(respectively),*
- $x_1, \ldots, x_k$ *are nullary functions defined over* $\mathbb{N}$ *and the only dynamic functions of the ASM; as usual we can identify them with variables, further denoted by* $Var(A)$*. The sequences* $in(A)$ *and* $out(A)$ *only contain such variables (*$in(A)$ *(resp.* $out(A)$*) denotes the inputs (resp. the outputs) of the algorithm).*

And, since static functions in an ASM are evaluated for free, when considering complexity, the choice of basic ASMs is not so restricted as it could seem on first view.

Remark 2 (The *min* example)**.** *A natural arithmetical ASM for the min problem is:*

A `inf` *algorithm* :
`if` $x = 0$ `then`
  $res := n$
`if` $\neg(x = 0) \wedge y = 0$ `then`
  $res := m$
`if` $\neg(x = 0) \wedge \neg(y = 0)$ `then`
  $x := x - 1$
  $y := y - 1$

Definition 7 (Arithmetical Primitive Recursive Algorithms)**.** *The set of* arithmetical primitive recursive algorithms *(*Apra*) is the set of couple* (A, f) *where* A *is a basic ASM and* $f \in \mathcal{PR}$*.* f *is a maximal bound for the length of the runs of* A*.*

Remark 3. *For instance, we can define* (A_{Ack}, min^2) *to be an* Apra *(*$A_{Ack}(n, m)$ *is an ASM (basic) computing the Ackermann function on* n, m *and we run it at most* $min^2(n, m)$*).*

4.4 Programming languages

4.4.1 A imperative programming language for Apra : Loop_{halt}

We consider the language $Loop^{\omega}$ (described in 2) but we do not allow high-order procedural variables. We add the following command:

$$c \ ::= \ \ldots \ | \ \texttt{halt;}$$

The semantics is just to force the halt of the program and this command does not allow to leave the set of primitive recursive functions.

We then proceed in two step for the Apra (A, f) with π_A for the program of A:

1. the π_A ASM program is translated into a LOOP_{halt}: the shell program,
2. as the function f is primitive recursive there exist a program in LOOP that implements it (at least intentionally) : the core program. We then "inject" the shell program into the core program.

The core program The core program is constructed from the π program by introducing new variables (that represent the state before the update) and by comparing old and new values to detect a fixed point.

Proposition 1. *Let A be a ASM and X, Y two states of $S(A)$ with $\pi_A(X) = Y$. Then, there exists a LOOP_{halt} program P_A such that:*

$$(P_A, \mu) \rightarrow^l (\epsilon, \mu')$$

for some l and some environments μ, μ' with $env(X) \subseteq \mu$ and $env(Y) \subseteq \mu'$ ($env(X)$ is the 0-ary variables of the state). Moreover, l is bounded independently from the inputs value of A.

Remark 4. *The number of variables are twice the number of variables of the ASM and there is no loop in the program (coming from a no-loop π program).*

The most interesting process is the construction of a program that has to compute sufficiently many times the core program.

The shell program To compute the core program we have two solutions:

1. first compute the f function and then construct a bounded loop with the core program in. BUT the way we compute f may be wrong (from a complexity point of view) : we know that some functions may not have a *good* implementation in LOOP. Then the computation of f may be too long compared to the run of the ASM.
2. benefit of the `halt;` command. Then to be sure that de core program is iterated enough, one can execute the core program each time a command of the shell program is executed. The `halt;` command allowing to stop at the right moment.

Definition 8. *Let P be a LOOP program and Q be a LOOP_{halt} program. We define the* **insertion** *of Q in P (denoted by $P[Q]$) by induction on the length of P:*

- *if P is $x := e;\ com_s$ then*

$$P[Q] \text{ is } Q\ x := e;\ com_s[Q]$$

- *if P is* `if` e `then` com_1 `else` com_2 `endif`; com_s *then*

$$P[Q] \text{ is } Q\ \texttt{if}\ e\ \texttt{then}\ com_1[Q]\ \texttt{else}\ com_2[Q]\ \texttt{endif};\ com_s[Q]$$

- *if P is* `loop` $expr_{int}$ `do` com `endloop`; com_s *then*

$$P[Q] \text{ is } Q\ \texttt{loop}\ expr_{int}\ \texttt{do}\ com[Q]\ Q\ \texttt{endloop};\ com_s[Q]$$

Theorem 3. *Let g be a function from $\mathbb{N}^k$ to $\mathbb{N}$ and (A, f) be a APRA which computes g. Then there exists a LOOP_{halt} program that simulates A whith a complexity belongs to $O(f)$.*

This leads to consider the escape of a bounded loop as a good control structure for programming language. Of course, one can introduce more flexible control structure such that exception or more subtle block escaping (see [ADA02] for instance).

4.4.2 A functional programming language for APRA

Following [CLV06] and [CPV09], we extract from the system T of Gödel, the core of the functional language that simulates the class $APRA$. The fragment of system T which captures $APRA$ is exactly the language induced by the definition of *primitive recursion with variable parameters* (see [Pét68]) with a mixed call-by strategy (by value on β–reduction and by-name for `rec` and `if` reductions).

But the simulation relies on another control structure related to the bounded loop itself that can be called *conditional bounded loop*. This kind of control structure has been used in PL/1. For sake of simplicity we consider also *down-to* bounded loops : the loop go downto 1.

Definition 9.

$$\textit{command} \quad ::= \quad ... \mid \texttt{for}\ x_i\ \texttt{in}\ b'\ \texttt{downto}\ 1\ \texttt{onlyIf}\ b\ \{s\}$$

4.4.3 Lock-step simulation

In order to distinguish the successor S (which is a constructor) from the successor seen as an operation (whose evaluation should imply a reduction step), we shall use the keyword `succ` as an abbreviation for $\lambda x.S(x)$ in the following definition:

Definition 10. *The translation* $^+$ *of* LoopC *program with variables* $\vec{x} = (x_1, \ldots, x_n)$ *into a term is defined by induction on expression and command as follows:*

- $\overline{n}^+ = S^n(0)$
- $\overline{\texttt{true}}^+ = \texttt{true}$ *and* $\overline{\texttt{false}}^+ = \texttt{false}$
- $x_i^+ = x_i$
- $(x_i + 1)^+ = \texttt{succ}(x_i)$
- $(x_i - 1)^+ = \texttt{pred}(x_i)$
- $\{\}^+ = \vec{x}$
- $\{x_i = e;\ s\}^+ = \texttt{let}\ x_i\ =\ e^+\ \texttt{in}\ \{s\}^+$
- $\{c;\ s\}^+ = \texttt{let}\ \vec{x}\ =\ c^+\ \texttt{in}\ \{s\}^+$ *if* c *is not an assignment*
- $(\texttt{if}\ b\ \{s_1\}\ \texttt{else}\ \{s_2\})^+ = \texttt{if}(b^+, \{s_1\}^+, \{s_2\}^+)$
- $(\texttt{for}\ x_i\ \texttt{in}\ e\ \texttt{downto}\ 1\ \texttt{onlyIf}\ b\ \{s\})^+ =$

$$\texttt{rec}_{N\to N}(e^+, \lambda\vec{x}.\vec{x},\ [y, x_i]\lambda\vec{x}.\texttt{if}(b^+, \texttt{let}\ \vec{x} = \{s\}^+\ \texttt{in}\ y\ \vec{x},\ \vec{x}))\ \ \vec{x}$$

Theorem 4. *If* $\langle c, (\vec{x}, \vec{n})\rangle \rightsquigarrow \langle c_1, (\vec{x}, \vec{n_1})\rangle$ *then* $c^+[\vec{n}^+/\vec{x}] \twoheadrightarrow c_1^+[\vec{n_1}^+/\vec{x}]$.

The simulation is not strict : one step in the loop is translated to at most two steps in the *rec* term.

In fact the obtained term is writable in PRV. Then we answer a question asked in[Val05] where we compared call-by-value and call-by-name in T_0 and proposed a storage operator in PRV better than that of T_0.

4.5 Conclusion and perspectives

Abstract State Machines are a powerful tool for the study of class of algorithms rather than class of functions. The distinction between the core of the algorithms (on which one may focus) and the nutshell (which is an evaluation of how many times the core must be run before a *possibly* interesting result) opens a new way to consider algorithms.

The approach (limited to primitive recursive algorithms) presented in this chapter is devoted to study the expressiveness of a programming language from a complexity point of view, linking theory of algorithms and theory of computable functions. The set of algorithms that one can implement in a programming language is the composition of a core program (center on some simple control structures: conditional, escape, ... without loops) and the possibility to implement the complexity of the run.

The set $APRA$ of algorithms is large enough to include well known *tractable* algorithmic problems. The two programming languages that are used to implement those algorithms are closed to usual language (Loop in the imperative paradigm and recursion with variable parameters for the functional one).

What next? The distinction between the core of algorithm and the halting conditions might be of an interesting approach. Many directions might be worth considering

1. other model of algorithms like Moschovakis model's,
2. the halting condition may be a logic formula that can be constructively proved for instance,
3. others classes of algorithms : polynomials, elementary, ...

Chapter 5

Evolving MultiAlgebras

5.1 Context

Abstract State Machine (described in section 4.2) has been developed since 1985 at least in two directions:

- considering ASM as a specification language : almost all computation models have been described within this framework, a large set of usual algorithms has been formalized too. This branch has been popularized first by Y. Gurevich and now essentially by E. Bŕoger : see the web site `http://www.eecs.umich.edu/gasm/`
- considering ASM as a formalization of the theoretical notion of *algorithms*: this part is mostly supported by Y. Gurevich and people around him (see paper on `http://research.microsoft.com/en-us/um/people/gurevich/annotated.htm`. The foundations have been enforced and fundamental results have been proven.

The fact that Abstract State Machines (ASMs) can strict lock-step (i.e.*step-by-step*) simulate any kind of machines (Turing machines, stack automata, RAM, etc) and grammars was shown long ago by Gurevich (in [DDG97], [Gur93]). A systematic study is also done in Börger [BS03]. A tighter notion of simulation is also valid as shown in Blass, Dershowitz & Gurevich [ABG09].

Up to now, the approach of simulation succeed the following schema: the simulated model is embedded in the state of the ASM and the program of the ASM is then an interpreter/simulator of the model. The Fig. 5.1 illustrates the schema.

This approach neglects the definition of the model itself and usually adds some wart (*verrue* in french) to the state (useful for the simulation). Fig.5.2 illustrates this point of view for the Turing machine model.

We consider that a better approach at most for models of computation is to consider that the simulated program **must be** the simulator program rather than a data. This leads to the representation of the Fig.5.3 and opens new questions.

The questions we consider in this chapter are the following:

(Q1) Can we replace strict lock-step simulation by literal identity (up to a simple change of view)?

(Q2) Given a computation model $\mathcal{C}$, is it possible to get a natural characterization of the class of ASMs which are equivalent to machines in $\mathcal{C}$?

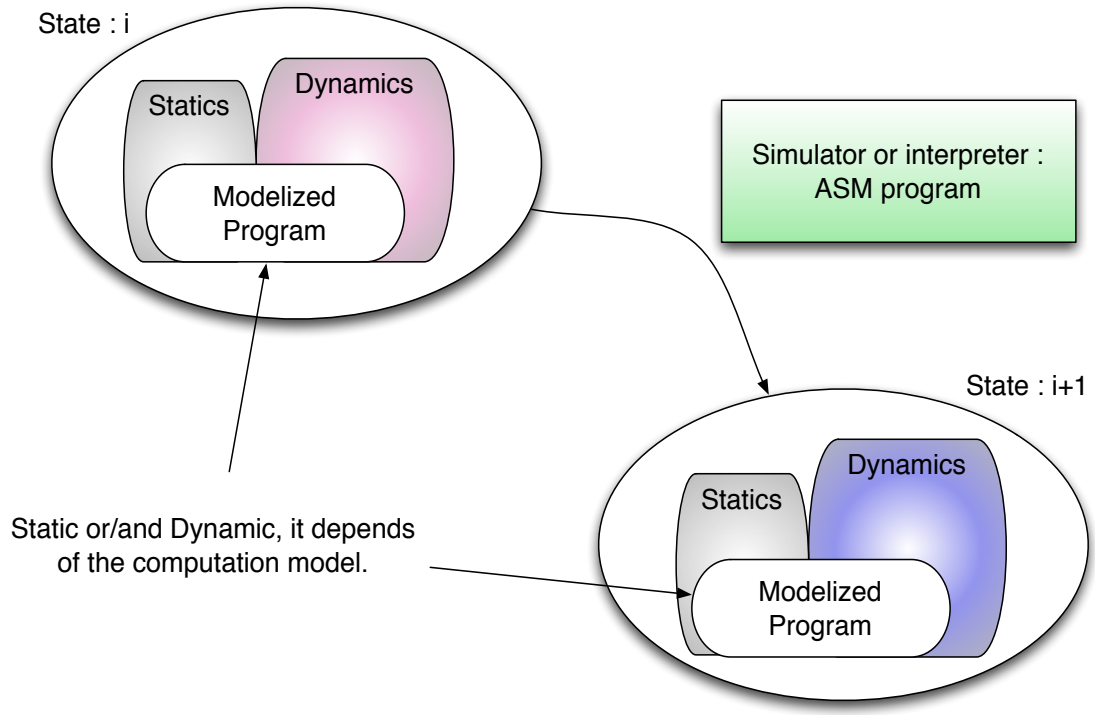

Figure 5.1: Pattern of model computation simulation

As far as we know, up to now, there is only one isolated answer which is about question (Q2): Gurevich & al. [DDG97] proved that Schönhage Storage Modification Machines correspond exactly (for strict lock-step equivalence) to ASMs with unary functions only.
We bring positive answers to both questions for the diverse usual computation models $\mathcal{C}$ (Turing machines, stack automata, RAMs, Schönhage Machines, Chomsky type 0 grammars, etc.) slightly extended to models $\mathcal{C}^+$ using a tailored version of ASMs which (resurrecting Gurevich's original name for ASMs) we call *Evolving Multialgebras* (EMAs). These answers have the following remarkably simple form:

Theorem 5. *There exists a family of EMA static parts $\mathcal{M}$ (fixed semantical feature) and a family of dynamic signatures $\mathcal{S}$ (fixed syntactical feature) such that, letting $\mathcal{E}_{M,S}$ be the family of EMAs with static part in $\mathcal{M}$ and dynamic signature in $\mathcal{S}$,*
- any computation device in $\mathcal{C}^+$ is literally identical to some EMA in $\mathcal{E}_{\mathcal{M},\mathcal{S}}$,
- this "literal identity" correspondence is a bijection from $\mathcal{C}^+$ onto $\mathcal{E}_{\mathcal{M},\mathcal{S}}$.

Of course, *literal identity* is not a formal notion. What we mean is as follows: the diverse components of a computation device in $\mathcal{C}^+$ are in one-one correspondance with the diverse components of the associated EMA, and this correspondance is an identity up to a change of perspective (for instance, a "physical" bi-infinite tape will be considered to be identical to the mathematical set $\mathbb{Z}$ of integers).

Remark 5. *1. This theorem is indeed a schema: one theorem per computation model. We have proved it for a variety of usual sequential computation models (cf. [GV10]).*

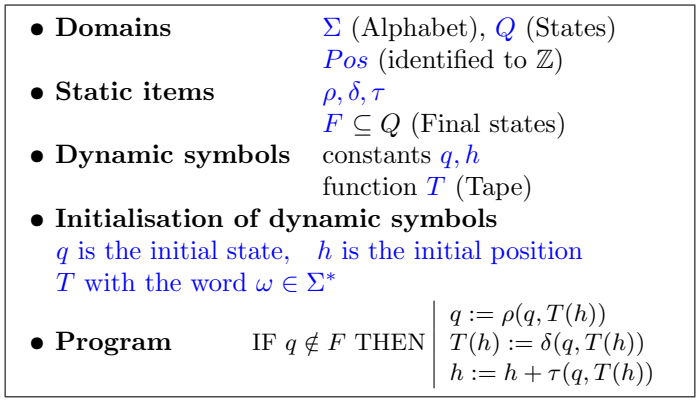

Figure 5.2: Turing Machine via an ASM

2. As said above, the diverse instances of Theorem 5 are proved for slight extension $\mathcal{C}^+$ of the usual computation models $\mathcal{C}$. In all cases, $\mathcal{C}^+$ can be viewed as $\mathcal{C}$ considered with different time units: for any $k \geq 1$, a device $\mathcal{M}$ in $\mathcal{C}$ is seen as a device $\mathcal{M}^{(k)}$ in $\mathcal{C}^+$ in which one step of $\mathcal{M}^{(k)}$ corresponds to k successive steps of $\mathcal{M}$ (or $< k$ successive steps in case the last of these steps has no successor).
3. Considering another presentation of $\mathcal{C}^+$, one can also view it as $\mathcal{C}$ in which some contingencies have been removed (for instance, the read/write head will be able to scan a window of cells instead of a single cell) but the computational paradigm has been preserved: local computation and a particular topology of data storage for Turing machines, indirect addressing of registers for random access machines, etc. In our opinion, the classes $\mathcal{C}^+$ are the right ones *to carry the diverse computation paradigms.*
4. In fact, contingencies can also be captured by families of EMAs with more technical definitions (cf. [GV10]): we loose the remarkable simplicity of the above families $\mathcal{E}_{\mathcal{M},\mathcal{S}}$.
5. This theorem schema strengthens Gurevich's claim that ASMs constitute the natural mathematical modelization of algorithms: EMAs (which are a variant of ASMs) appear as the computation model which unifies all usual sequential computation paradigms.

5.2 Evolving Multi-Algebras

Evolving MultiAlgebras are modified ASM in the following sense (we illustrate difference with Turing machine model):

- rather than to consider a whole algebra, it's useful (and even more accurate) to consider the typing of sets and functions on that sets we used. So, many algebras are defined as static or dynamic symbols. By this way, we are closed to good practice in programming, and also we forbid the shuffle of some operations.

- more than typing we need to consider multicopy of some sets and functions. *For instance, if we want to represent a multi-tape Turing machine we need to represent each tapes independently (same for functions operating on them).*

- halting conditions are outsourced. As in Ch.4, where we consider the length of the run as a function outsourced, we give halting conditions in the meta-level. *The halting condition*

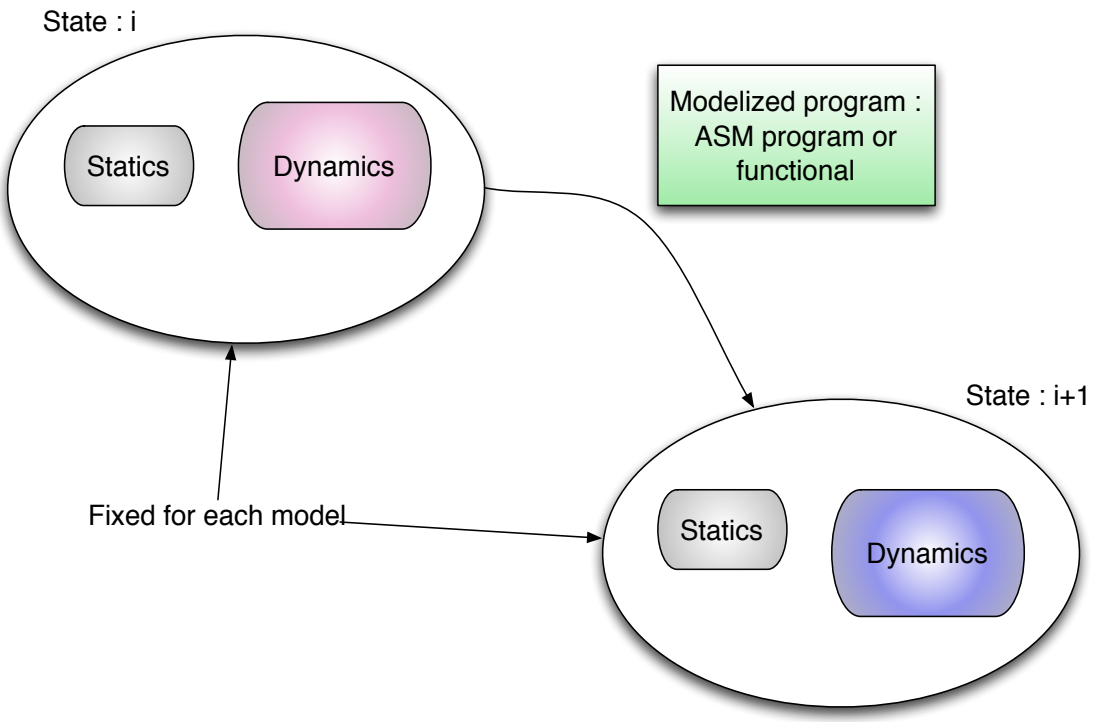

Figure 5.3: More interesting model computation simulation

in a Turing machine is that a certain state (belonging to a final states set) occurs during the computation.

- the program is no more considered as a set of updates guarded by some formula, instead of we consider a transition functional which does exactly what the program tells to do, but want to be more semantics than operational. *An ASM program mimicking the transition function δ of a Turing machine is a description of δ. Since there are many distinct descriptions of the same δ, there are many ASMs which tightly simulate the same Turing machine. Thus, surprisingly as it is, as concerns the transition function component, ASMs are less abstract than Turing machines. Somehow, there is an extra operational feature in ASMs which has no corresponding part in Turing machines: the operational way to use δ is not part of the formalization of Turing machines. Which precludes any literal identity in ASM simulation of Turing machines.*

No surprise, the proof of Theorem 5 for a particular $\mathcal{C}$ involves the particular features of the class $\mathcal{C}$. Thus, the claim (point 5 in Remark 5) that Theorem 5 is true for extensions $\mathcal{C}^+$ of every usual sequential computation model $\mathcal{C}$ cannot be proved but only be supported by proved instances for a variety of classes $\mathcal{C}$.

As for the common features to all such proofs, they come from an analysis of what precludes positive solutions to questions (Q1) and (Q2). Let us list some of the difficulties which are met. Some are easy to solve, other ones force to adequately tailor the definition of ASMs (as that of EMAs) and those of the usual computation models.

(1) Again considering Turing machines, an ASM simulates the tape by the set $\mathbb{Z}$ of all integers and the moves of the head by the successor and predecessor operations on $\mathbb{Z}$. Terms in the ASM logical language allow to name the i-th successor and the i-th predecessor. Thus, we cannot avoid the ASM program to move the head more than one cell left or right unless we constrain terms in ASM programs to be of a simple form (somewhat "flat"). Which would put technicalities to any positive answer to question (Q2). This is why we consider slight extensions of the machine models which allow the read/write head to scan a window of cells rather than only one cell and to move in a window. This is a kind of extra capability which is much like allowing several tapes or several heads. Though it does modify the model, it does preserves its core feature: successive local actions.

(2) For machine models having programs like RAMs and SMM (Schönhage Storage Modication Machines), there are two slight modifications. First, allow bounded blocks of parallel and/or successive actions. Second, remove the program and the program counter in favor of a transition function (much in the vein of Turing machines) which, though operating on an infinite set (the contents of the accumulator and of the addressed registers in the case of RAMs) is very simply definable in terms of the original program. Thus, we replace an operational item (the program) by a denotational one (the transition function). Again, though it does modify the model, it does preserves its core feature: indirect addressing (for RAMs), dynamic storage topology (for Schönhage pointer machines).

EMAs versus ASMs. In our opinion, ASMs and EMAs are complementary models. EMAs generalize any type of machine: it is the unification model. As for ASMs, they are closer to programming. Indeed, the functioning of a EMA goes through the iteration of a functional. To program an EMA, we need to add some operational information on how to use this functional and this leads back to a program, hence to an ASM... Thus, ASMs are EMAs plus the instructions for using the functional: ASMs refine EMAs (in the sense of software engineering) and EMAs are a (more) abstract version of ASMs.

5.2.1 About the transitional functional

We give here the intuition of this functional by comparing with the Turing transition function.

Usually the transition function of a Turing machine is typed as follows:

$$\delta : Q \times \Sigma \to Q \times \Sigma \times \{-1, 0, +1\}$$

Q stands for the set of states and Σ stands for the set of symbols. It is presented by rules like the following:

$$\begin{array}{rcl} \delta(q_1, 1) & = & (q_1, 1, +1) \\ \delta(q_1, 0) & = & (q_2, 1, \ \ 0) \\ \delta(q_2, 1) & = & (q_2, 1, -1) \end{array}$$

The operational semantics is described informally (for the first rule):

> **if** the current state *is* q_1 and the cell read *is* 1 **then** the current state *is now* q_1, 1 *is now written* on the current cell, the head *moves now* to the right.
>
> There is also a meta-rule where the case of final state is considered.

This can be described by the following ASM program (for the three rules and if s is a symbolics symbol for the current state, *pos* for the position, w for the tape):

```
if s = q1 ∧ w(pos) = 1 then
    s := q1
    w(pos) := 1
    pos := pos + 1
if s = q1 ∧ w(pos) = 0 then
    s := q2
    w(pos) := 1
    pos := pos
if s = q2 ∧ w(pos) = 1 then
    s := q2
    w(pos) := 1
    pos := pos − 1
```

For sake of simplicity, in this program we do not take into account the meta-rule (the current state is or is not a final state).

From this example, one can generalized the construction of the functional then it follows that the transitional functional takes as parameter a set of values (of static and dynamics symbols) to be compared and gives as output a set of values (to update dynamics symbols): thanks to the typing that manages the operational semantics (see [GV10] for formal presentation).

5.3 The Turing machine sample

The identification of Turing machines with particular EMAs given in this section leads to also consider a slight variant of Turing machines, which we call "window Turing machines":

- the head is allowed to scan a small window instead of a single cell, and to move inside a window in a single step,
- halting (be it accepting or rejecting) is not related to the current state but to the current local configuration: the state plus the contents of the scanned window.

5.3.1 Deterministic window Turing machines

Definition 11. *A deterministic k-window n-tape (bi-infinite tapes) Turing machine is a tuple*

$$(n, k, \Sigma, Q, F^+, F^-, \delta, \omega_i, \mu_i)_{i=1,\dots,n}$$

where, for $i = 1, \dots, n$,

- *Σ and Q are disjoint finite sets (the alphabet and the set of states),*
- *$F^+, F^- \subseteq Q \times \Sigma^{(n2k+1)}$ are disjoint sets*
  (accepting/rejecting final local configurations),
- *$\delta : Q \times \Sigma^{n(2k+1)} \to Q$* *(state transition),*
- *$\tau_i : Q \times \Sigma^{n(2k+1)} \to \Sigma^{n(2k+1)}$* *(read/write on tape i),*
- *$\mu_i : Q \times \Sigma^{n(2k+1)} \to \{-k-1, \dots, -1, 0, 1, \dots, k+1\}$* *(move on tape i).*

On each tape, the head scans the cell on which it is positioned and the k cells to the left and the k cells to the right, (a total of $2k+1$ cells). The argument of type $\Sigma^{n(2k+1)}$ in δ, ω_i, μ_i is the contents of the $n(2k+1)$ cells scanned on the n tapes. The effect of a transition is to change the state according to δ, to modify the contents of the scanned cells of tape i according to τ_i and to move its head according to μ_i.
The notions of run, halt, acceptance and rejection are defined as usual.

Remark 6. *Usual deterministic n-tape Turing machines are the 0-window ones.*

5.3.2 EMAs and deterministic n-tape Turing machines

Definition 12. *We denote by $\mathcal{C}_{wT}^{(n)}$ the class of EMAs*

$$\mathcal{A} = (n+3;\ \mathcal{S}_{sta}^{int}, \mathcal{S}_{sta}^{ext}, \mathcal{S}_{dyn}^{int};\ \mathcal{D};\ \mathcal{M};\ \Phi)$$

which satisfy the following conditions (for clarity, we give the semantic types symbols and abusively denote by the same letter static symbols and their interpretations in the structure $\mathcal{M}$ of the EMA).

(1) $\mathcal{A}$ has $n+3$ sorts and its multidomain is

$$\mathcal{D} = (\mathbb{Z}^{(1)}, \ldots, \mathbb{Z}^{(n)}, Q, \Sigma, \mathfrak{S})$$

where the $\mathbb{Z}^{(i)}$'s are pairwise disjoint copies of $\mathbb{Z}$ (for instance, $\mathbb{Z}^{(i)} = \mathbb{Z} \times \{i\}$), Q, Σ are finite sets and $\mathfrak{S} = \{go, acc, rej\}$.

(2) The signature $\mathcal{S}_{sta}^{int}$ (for the static framework) contains:

- *two unary functions symbols $Succ^{(i)}, Pred^{(i)}$ of (semantic) type $\mathbb{Z}^{(i)} \to \mathbb{Z}^{(i)}$, for each $i = 1, \ldots, n$,*
- *$|Q|$ constants $(q_r)_{r \in Q}$ of type Q, $|\Sigma|$ constants $(\sigma_s)_{s \in \Sigma}$ of type Σ and three constants go, acc, rej of type $\mathfrak{S}$.*

(3) The signature $\mathcal{S}_{sta}^{ext}$ (for static symbols) is empty.

(4) The signature $\mathcal{S}_{dyn}^{int}$ (for the dynamic environment) contains:

- *n constants $pos^{(1)}, \ldots, pos^{(n)}$ of respective types $\mathbb{Z}^{(1)}, \ldots, \mathbb{Z}^{(n)}$, one constant q of type Q, and one constant $\mathfrak{s}$ of type $\mathfrak{S}$,*
- *n unary functions $c^{(1)}, \ldots, c^{(n)}$ of respective types $\mathbb{Z}^{(1)} \to \Sigma, \ldots, \mathbb{Z}^{(n)} \to \Sigma$.*

(5) The interpretations in $\mathcal{M}$ of the symbols of $\mathcal{S}_{sta}^{int}$ (that is, the static framework) are as follows:

- *for each $i = 1, \ldots, n$, $Succ^{(i)}, Pred^{(i)}$ are interpreted as the successor and predecessor functions in the copy $\mathbb{Z}^{(i)}$ of $\mathbb{Z}$,*
- *$q_0, \ldots, q_{r-1}$ are interpreted as the r elements of Q,*
  $\sigma_0, \ldots, \sigma_{s-1}$ are interpreted as the s elements of Σ,
  and go, acc, rej are interpreted as the 3 elements of $\mathfrak{S}$.

Thus, the EMAs in $\mathcal{C}_T^{(n)}$ are defined as those having particular signature, multidomain and fixed static framework, parametered by two integers $r, s \geq 1$. In particular, there is no condition on the functional Φ.

Theorem 6 (EMA representation theorem for Turing machines [GV10])**.** *Any deterministic n-tape window Turing machine is* literally identical *to some EMA in the class $\mathcal{C}_T^{(n)}$. Conversely, any EMA in $\mathcal{C}_T^{(n)}$ is* literally identical *to some deterministic n-tape window Turing machine.*

Identification The argument is based on the following literal identifications between the components of a Turing machine (TM) and the interpretations of symbols of the EMA signature:

1. (TM) i-th tape and the way the read/write head moves on it.
   (EMA) the copy $\mathbb{Z}^{(i)}$ of $\mathbb{Z}$ structured as $\langle \mathbb{Z}^{(i)}, Succ^{(i)}, Pred^{(i)} \rangle$.

2. (TM) states and letters.
   (EMA) interpretations of the static symbols $q_0, \ldots, q_{r-1}$ and $\sigma_0, \ldots, \sigma_{s-1}$.

3. (TM) current state, positions of the n heads and contents of the n tapes.
   (EMA) current interpretations of the dynamic symbols q, $pos^{(1)}$,...,$pos^{(n)}$ and $c^{(1)}, \ldots, c^{(n)}$.

4. (TM) non final or final accepting/rejecting character of the current state.
   (EMA) current interpretation of the dynamic symbol $\mathfrak{s}$.

5. (TM) transition function.
   (EMA) local semialgebraic functional.

5.4 Conclusion and perspectives

The process to enforce the program, to be evaluated, to be the ASM program (and no more in the state) is a fundamental new perspective for the simulation of models of computation. We have improved ASM definition, in the way (essentially) of typing and adding multi-domain. It has lead us to identify usual models of computation to some EMAs.

We then obtain a characterization of models of computation by some EMAs (static and dynamics symbols), letting the program free of constraints.

The result has, form my point of view, some perspectives:

- one can reconsider the definition of models of computation and enlarge them to be the one defined by their EMAs definition; the windows Turing machine model is a convincing example.

- as we can construct the set of algorithms of models of computation in the same framework: can we compare those sets?

- what about programming language models; ASMs are very closed to imperative paradigm and models of computation that have finite updates seem to be easily characterizable by some EMAs but what about functional or logical languages? The closure in programming languages are a powerful tool to delay the evaluation of some expressions. It allows to modify the environment with a kind of infinite changes.

Those questions are of some interests for me and some of those questions are closed to be answered in next works.

Bibliography

[ABG09] Nachum Derschowitz Andreas Blass and Yuri Gurevich, *Exact exploration*, Tech. Report MSR-TR-2009-99, Microsoft Research, July 2009.

[ADA02] *Consolidated ada reference manual: language and standard libraries*, Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[APV] Ph. Andary, B. Patrou, and P. Valarcher, *A theorem of representation for primitive recursive algorithms*, Fundamenta Informaticae **to appear**.

[BD95] Stephen Brookes and Denis Dancanet, *Sequential algorithms, deterministic parallelism, and intensional expressiveness*, 22nd Annual Symposium on POPL, 1995.

[BS03] E. Börger and R. Stärk, *Abstract state machines: a method for high-level system design and analysis*, Springer-Verlag, 2003.

[BW00] A. Beckmann and A. Weiermann, *Characterizing the elementary recursive functions by a fragment of Godel's T*, Archive for Mathematical Logic **39** (2000), 475–491.

[CF98] L. Colson and D. Fredholm, *System t, call-by-value and the minimum problem*, Theoretical Computer Science **206** (1998), 301–315.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, second edition ed., no. pg.902, MIT Press and McGraw-Hill, 2001.

[CLV06] T. Crolard, S. Lacas, and P. Valarcher, *On the expressive power of loop language*, Nordic Journal of Computing **13** (2006), 46–57.

[Col89] L. Colson, *About primitive recursive algorithms*, Theoretical Computer Science **372** (1989), 194–206.

[Col91] ———, *Représentation intensionnelle d'algorithmes dans les systèmes fonctionnels*, Thèse de doctorat, Université Paris 7 (1991).

[Col96] ———, *A unary representation result for system t*, Annals of Mathematics and Artificial Intelligence **16:385-403** (1996).

[Coq92] T. Coquand, *Une preuve directe du thérorème d'ultime obstination*, Compte Rendus de l'Académie des Sc. **314, Serie I** (1992), 389–392.

[CP09] T. Crolard and E. Polonowski, *A program logic for higher-order procedural variables and non-local jumps*, Tech. Report TR-LACL-2009-3, Université Paris Est, 2009.

[CPV09] T. Crolard, E. Polonowski, and P. Valarcher, *Extending the loop language with higher-order procedural variables*, ACM TOCL **10** (2009), no. 4, 1–36.

[Dav93] R. David, *Un algorithme primitif récursif pour la fonction inf*, Compt. Rend. de l'Aca. des Scie. **317 (Série I)** (1993).

[Dav94] _____, *The inf function in the system f*, Theoretical Computer Science **135:423-431** (1994).

[Dav97] _____, *On the asymptotic behaviour of primitive recursive algorithms (part 2)*, Submitted to TCS (1997).

[Dav01] _____, *On the asymptotic behaviour of primitive recursive algorithms*, Theor. Comput. Sci. **266** (2001), 159–193.

[Dav03] R. David, *Decidability results for primitive recursive algorithms*, Theoretical Computer Science **300** (2003), 477–504.

[DB96] D. Dancanet and S. Brookes, *Programming language expressiveness and circuit complexity*, Internat. Conf. on the Mathematical Foundations of Programming Semantics, 1996.

[DDG97] S. Dexter, P. Doyle, and Y. Gurevich, *Gurevich abstract state machine and schonhage storage modification machines*, J. Universal Computer Science **3** (1997), 279–303.

[Dea07] Walter Dean, *What algorithms could not be*, Ph.D. thesis, New Brunswick Rutgers, The State University of New Jersey, 2007.

[DG08] N. Dershowitz and Y. Gurevich, *A natural axiomatization of church's thesis*, Bulletin of symbolic logic **14** (2008), no. 3, 299–350.

[Dri03] L. Van Den Dries, *Generating the greatest common divisor, and limitations of primitive recursive algorithms*, to appear in Foundations of Computational Mathematics (2003).

[DV10] René David and Pierre Valarcher, *Trace of some primitive recursive schema*, Tech. Report TR-LACL-2010-3, LACL (Laboratory of Algorithms, Complexity and Logic), Université of Paris Est, 2010.

[DW83] M. Davis and E. Weyuker, *Computability, complexity and languages*, Academic Press, 1983.

[Esc93] Martin Hotzel Escardo, *On lazy natural numbers with applications*, SIGACT News **24** (1993), no. 1, 61–67.

[Fre96] D. Fredholm, *Computing minimum with primitive recursion over lists*, Theoretical Computer Science **163** (1996), 269–276.

[Gan80] Robin Gandy, *Church's thesis and principles for mechanisms*, The Kleene Symposium (Keisler H.J. Kunen K. Barwise, J., ed.), North-Holland, Amsterdam, 1980, pp. 123–148.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor, *Proofs and types*, Cambridge Tracts in Theoretical Computer Science, no. 7, Cambridge University Press, 1989.

[GMV10] F. Gervais, D. Michel, and P. Valarcher, *B-asm: Specification of asm à la b*, ABZ (Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, eds.), Lecture Notes in Computer Science, vol. 5977, Short paper, Springer, 2010, p. 391.

[Grz53] A. Grzegorczyk, *Some classes of recursive functions*, Rozprawy matematyczne (http://matwbn.icm.edu.pl/ksiazki/rm/rm04/rm0401.pdf), vol. 4, 1953, pp. 1–45.

[Gur85] Yuri Gurevich, *A new thesis*, Abstracts, American Mathematical Society, 1985.

[Gur93] ———, *Evolving Algebras 1993: Lipari Guide*, Specification and Validation Methods, Oxford University Press, 1993, pp. 9–36.

[Gur00] Y. Gurevich, *Sequential abstract state machines capture sequential algorithms*, ACM Transactions on Computational Logic **1** (2000), 77–111.

[GV10] S. Grigorieff and P. Valarcher, *Evolving multialgebras unify all usual sequential computation models*, STACS, 2010.

[Mos98] Y. N. Moschovakis, *On founding the theory of algorithms*, Truth in mathematics (H. G. Dales and G. Oliveri, eds.), Clarendon Press, Oxford, 1998, pp. 71–104.

[Mos01] ———, *What is an algorithm ?*, Mathematics unlimited – 2001 and beyond (Springer, ed.), B. Engquist and W. Schmid, 2001, pp. 919–936.

[Mos03] ———, *On primitive recursive algorithms and the greatest common divisor function*, Theor. Comput. Sci. **301** (2003), 1–30.

[MP08] Y. N. Moschovakis and V. Paschalis, *Elementary algorithms and their implementations*, New Computational Paradigms (Benedikt Lowe S. B. Cooper and Andrea Sorbi, eds.), Springer, 2008.

[MR76] A. R. Meyer and D. M. Ritchie, *The complexity of loop programs*, Proc. ACM Nat. Meeting, 1976.

[MV09] D. Michel and P. Valarcher, *A total functional programming language that computes apra*, Studies in Weak Arithmetic, CSLI Lecture Notes, vol. 196, Stanford, 2009.

[MVY03] J.-F. Michon, P. Valarcher, and J.-B. Yunès, *Hfe and bdds : A practical attempt at cryptanalysis*, Progress in Computer Science and Applied Logic (C. Xing Kegin Feng, H Niederreiter, ed.), Birkhauser Verlag, 2003.

[MVY05] ———, *On maximal qrobdd's boolean functions*, RAIRO T.I.A. **39** (2005), no. ITA0442, 677–786.

[Pét68] Rosza Péter, *Recursive functions*, Academic Press, 1968.

[Rei03a] W. Reisig, *On Gurevich's Theorem on Sequential Algorithms*, Acta Informatica **39** (2003), no. 5, 273–305.

[Rei03b] ———, *The Expressive Power of Abstract State Machines*, Computing and Informatics **22** (2003), no. 3, 209–219.

[Sch67] K. Schütte, *Proof theory*, Addison Wesley, 1967.

[Sch80] A. Schonhage, *Storage modification machines*, SIAM J. Comput. **9** (1980), 490–508.

[Sip05] M. Sipser, *Introduction to the theory of computation*, 2nd ed., Course Technology, 2005.

[Val96] P. Valarcher, *Intensionality vs extensionality and primitive recursion*, Lecture Notes in Computer Science (Spriger, ed.), vol. 1179, 1996, pp. 142–151.

[Val00] _____, *Intensional semantics of system t of godel*, Workshop on Domain 4 (ENTCS, ed.), vol. 35, 2000, p. 14.

[Val05] _____, *Call-by-value and call-by-name in primitive recursion : storage operator*, 5th International Workshop on Reduction Strategies in Rewriting and Programming, 2005.

[Val08] _____, *A complete characterization of primitive recursive intensional behaviours*, Theoretical Informatics and Aplications (2008), 69–82.

Chapter 6

Boolean functions: Cryptography and Applications

6.1 Introduction

My work on boolean functions began with the practical attempt to cryptanalysis of a public key encryption known as HFE Hidden Field Equation (see [14, 21, 22]).

A public key cryptosystem is a pair of functions (performed by an algorithm and denoted by f, g), which allows anyone to encrypt a message (denoted M) using a key known from all (public key denoted c). The encrypted message (denoted M') is obtained as following: $M' = F_C(M)$, where F_C is a function entirely determined by the function f and the c key. The decryption is done through the private key c' and function g we find $g_M = (c')(M')$.

The construction of a cryptosystem is to build these functions f and g and keys. We say that a cryptosystem is *sure* if it is difficult to found the message M from f, g and c. The notion of difficulty is often associated (for theoretical reasons) to the notion of the hard problem (or NP-hard) in computability theory (see [Sip05] example).

6.1.1 HFE

The HFE cryptosystems are constructed using polynomials defined over finite fields. The method can be described as follows:

Take a field $\mathbb{K}$ extension of degree n of $\mathbb{F}_2$. Then $\mathbb{K}$ can be seen as a vector space of dimension n on $\mathbb{F}_2$. Any basis $e_1,\ ldots, e_n$ of $\mathbb{K}$ defines a bijection of $\mathbb{K}$ to $(\mathbb{F}_2)^n$ as follows:

$$\sum_{i=1}^{n} x_i.e_i \longleftrightarrow (x_1, \ldots, x_n), \text{ with } x_i \in \mathbb{F}_2$$

Then every univariate quasi-quadratic polynomial

$$P(X) = \sum_{i,j} \alpha_{i,j} X^{2^i+2^j} + \gamma \text{ with } \alpha_{i,j} \in \mathbb{K} \text{ and } \gamma \in \mathbb{K}$$

on $\mathbb{K}$ can be seen in $(\mathbb{F}_2)^n$ as a set of multivariate quasi-quadratic polynomials on $\mathbb{F}_2$:

$$\left\{\begin{array}{rcll} P_1(x_1,\ldots,x_n) & = & \sum_{j,k}^{1..n} \beta_{1,j,k} x_i x_k + \delta_1 & \\ & \vdots & & \text{with } \beta_{i,j,k} \in \mathbb{F}_2 \text{ and } \delta_i \in \mathbb{F}_2 \\ P_n(x_1,\ldots,x_n) & = & \sum_{j,k}^{1..n} \beta_{n,j,k} x_i x_k + \delta_n & \end{array}\right.$$

Such a system is the public key of a system HFE. Encryption is a simple application of polynomials P_i on the bits of the plaintext. As for the decryption, it is supposed to be NP-hard (see [6, 12, 3, 11]). However we know that if the degree of $P(X)$ is not too high (see [25]) Berlekamp's algorithm (see [1, 13]) performs in reasonable time.

To ensure the strength, it introduces two *trap doors* which are just permutations of $\mathbb{K}$ and we call S and T (note that they are expressed on $\mathbb{K}$ as a polynomial of the form $\sum_i \alpha_i(X^{2^i})$ and $\mathbb{F}_2$ an invertible linear transformation). Then the HFE public key is defined by the expression on $\mathbb{E}_2$ the polynomial $T(P(S(X)))$ (which is quasi-quadratic). But now the polynomial $T(P(S(X)))$ has degree too high to expect using Berlekamp's algorithm for finding roots. The private key is then made of the data S and T, the polynomial can be made public.

6.1.2 Cryptanalysis attempt

Our offense (see [17]) is classified in the class ciphertext attacks only where it is assumed to have at its disposal only the public key and a copy of the ciphertext. We decided not to try to algebraic analysis despite the fact that the system is, but to feign ignorance, taking the system as it stands: a system of equations with boolean variables. For algebraic cryptanalysis we can refer to the work of Nicolas Courtois (see [3]), Aviad Kipnis and Adi Shamir (see [12]), Jean-Charles Faugere (see [4]) and Louis Granboulan, Antoine Joux and Jacques Stern (see [7]).

Our idea was to use BDDs to represent the equations and solve the system. The BDDs are a representation of boolean functions which can handle them efficiently.

Thus, given a ciphertext $y_1,\ ldots, y_n$ HFE system allows us to write that

$$\left\{\begin{array}{rcl} y_1 & = & P_1(x_1,\ldots,x_n) \\ & \vdots & \\ y_n & = & P_n(x_1,\ldots,x_n) \end{array}\right.$$

We construct, from this system, a Boolean function corresponding to a combination of polynomials. We just have to satisfy the formula (that is to say finding boolean $x_1,\ ldots, x_n$ which satisfy the formula):

$$F(x_1,\ldots,x_n) \stackrel{def}{=} \bigwedge_{i=1}^{n} (1 + y_i + P_i(x_1,\ldots,x_n))$$

to extract the message.

The construction of the function uses the BDD by constructing iteration

$$F_(i+1) = F_i \bigwedge (1 + P_i + y_i(x_1,\ldots,x_n))$$

where $F_0 = 1$ and $F = F_n$.

It was a complete failure, it was impossible even to represent a single equation of a system for usable HFE (80 boolean variables using the standard practice).

The failure is due to the *complexity* of boolean functions generated in a HFE system .

We decided to try to make smaller HFE systems to achieve complete cryptanalysis method in our attempt to understand the phenomena involved.

For this we make the two software; one to build HFE systems according to our convenience (see [18]), and the other to handle efficiently generated BDD (see [28]).

The size of generated BDDs by the resolution algorithm is represented in the FIG. 6.1. In this diagram different curves appear. Each represents a typical *behavior* of our algorithm for solving a HFE system produced from a particular finite field. These curves measure the size of the BDD of each generated formulas F_i.

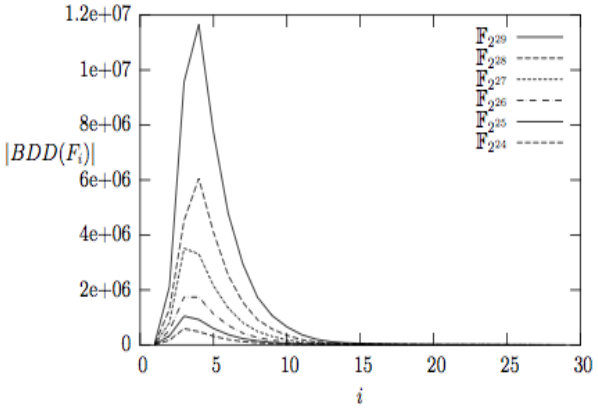

Figure 6.1: Size of BDDs generated by the algorithm of resolution.

Recall that the valuations that satisfy the formula F also satisfy any formula F_i. Why does a curve look like this? The thing to understand is that at each stage the number of valuations that satisfy the formula F_i decreases, but the *descriptive complexity* of this set of solutions changes depending on the curve. At first, the complexity strongly grows, then decreases sharply and collapses to a very small BDD. What we can observe is that the size of generated BDDs through a peak. We are interested in boolean functions corresponding to this peak and their representation within the BDD framework.

6.2 BDDs

The binary decision diagrams (BDD) are representations of boolean functions. For many useful functions, this representation is concise, allowing efficient manipulation of boolean functions through them. This structure is particularly used in the field of verification (see articles of Randal Bryant [2] and web site or Ingo Wegener [23, 24]).

The QROBDDs (Quasi-Reduced Ordered BDD) are canonical representations of boolean functions. A QROBDD is constructed as follows: starting from the truth table of a given boolean function f, we construct the binary tree which is canonically associated and whose leaves are labeled by `0` or `1`. While identifying all isomorphic subgraphs yields a directed acyclic graph which is called $QROBDD(f)$.

It should be noted that the QROBDD are a canonical representation of boolean functions not ROBDD. In fact, and for example, for any integer k, functions $f(x_1, \ldots, x_k) = 1$ have all the same ROBDD representation, but they have differents QROBDD representations.

The complexity $c_{\text{qrobdd}}(f)$ from such a representation checks the following trivial property:

$$n+1 \leq c_{\text{qrobdd}}(f) \leq 2^n + 1$$

If the lower bound is correct, this is not for the upper bound which value, denoted by $C(n)$, has been so much studied (see [8, 9, 10]). Our goal was then to describe all boolean functions with n variables for the maximum complexity. These functions are now denoted as *hard*–functions.

6.2.1 BDD profile

We can notice that a QROBDD is stepped, that it consists of nodes each located at some distance from the root. We call level i of a QROBDD, all nodes located at distance i of the root. We denote $\mathcal{P}(f)$ the QROBDD profile associated to f and $\mathcal{P}(f) = (r_0(f), \ldots, r_n(f))$ where $r_i(f)$ is the number of nodes in level i associated with the QROBDD f. The following relationships are known: $r_0(f) = 1$, $r_n(f) = 1$ or 2, $r_{i+1}(f) \leq 2.r_i(f)$ and $r_i(f) \leq r_{i+1}(f)^2$.

By definition we state

$$c_{\text{qrobdd}}(f) = \sum_{i=0}^{n} r_i(f)$$

What remains to be determined is how the nodes of a given level are connected to the nodes of the next stage?

To determine this we have established a constructive combinatorial result indicating that the number of ways to connect a stage of k nodes to a stage of m nodes and we note that (by analogy with the binomial coefficients) $C(m,k)$ satisfies the relation

1. $C(0,0) = 1$
2. if $k > m^2$ or $k < \frac{m}{2}$ then $C(m,k) = 0$
3. else

$$C(m,k) = \binom{m^2}{k} - \sum_{j=1}^{m} \binom{m}{j} C(m-j,k) > 0$$

or with a Pascal triangle shape

1. $C(0,0) = 1$, $\forall\, p > 0$, $C(0,p) = C(p,0) = 0$,
2. $C(1,1) = 1$, $\forall\, p > 1$, $C(1,p) = 0$,
3. $\forall\, m > 1$, $\forall\, k > 0$, then

$$\begin{aligned} kC(k,m) = {} & (m^2 - k + 1)C(k-1,m) \\ & + m(2m-1)C(k-1,m-1) \\ & + m(m-1)\ C(k-1,m-2) \end{aligned}$$

Then, the number of BDD with a profile $(r_0, \ldots, r_n)$ equals to $\prod_{i=1}^{n} C(r_i, r_{i-1})$.

What we have not yet been determined for a given complexity is, what are the possible profiles and hence the number of QROBDDs (of functions) with this complexity.

For particular values of the profiles, the number of BDD can be easily identified and associated functions characterized. Especially for BDDs of maximum size.

Now what connections exist between these graphs and boolean functions? This link is not completely clear, but if we fix a boolean functions decomposition (such as the Shannon one), then we associate to a given graph a particular boolean function. So by setting the Shannon decomposition (traditionally used to obtain a BDD), the question is to determine the set of boolean functions whose Shannon decomposition gives a boolean graph of maximum size. But the question might arise for other decompositions.

6.3 Hard boolean functions

For the Shannon decomposition, the result we obtained, is that hard boolean functions are all related to the Storage Access function (noted SA) which is the simplest amang the set of hard functions. If we noted $\mathcal{H}_n$ the set of hard functions with n variables, then for some values of n (when $n = a + 2^a$, for an integer a), the set $\mathcal{H}_n$ is the set of Twisted-SA; the set of SA functions *disrupted* (disturbed) by action of a symmetric group.

The function SA_k, for all $k = 2^a$, is a function with $n = a + 2^a$ variables and is defined by

$$SA_k(x_0, \ldots, x_{2^a-1}, y_0, \ldots, y_{a-1}) = x_m$$

where m is represented by $y_0, \ldots, y_{a-1}$ (in base two). It can be interpreted as a function that represents the memory access, the bit x_i is the memory state, and y_i of the memory address of the desired content.

The SA function is known to have a BDD representation with a maximum size (see [24]). This is not always the case; for some variables ordering we can obtained a polynomial size. But for our perspective changing the order of variables it's changing the function. Of course the functions are linked but they are not the same function.

Then we define the Σ-twisted SA function, for $\Sigma \in \mathfrak{S}_{2^k}$ permutation of 2^k elements. This induces a bijection on the set of k–uplets of bit for the representation of integer between 0 and $2^k - 1$, then

$$SA_k^{\Sigma}(x_0, \ldots, x_{2^a-1}, y_0, \ldots, y_{a-1}) = SA_k(\Sigma(X), y_0, \ldots, y_{a-1})$$

where X the integer with the binary representation $x_0 \cdots x_{k-1}$.

We first show that functions SA_k are hard. Then we show that $\mathcal{H}_{a+2^a} = \{SA_{2^a}^{\Sigma}\}$.

The other result is that for non-special values of n ($a + 2^a < n < a + 1 + 2^{a+1}$) the set of hard functions with n variables is obtained by injections from some SA_{2^a} and some $SA_{2^{a+1}}$.

These results are based on finding that the single degree of freedom existing in maximum QROBDD lies in the connections that appear in the inflection (stage at which the number of nodes decreases), and that it's in this place that acts the symmetric group.

Interesting properties were observed. For example there are hard functions that do not depend from all their variables. It is quite surprising because counterintuitive. To learn more, we refer the reader to read [16].

6.4 Conclusion

The attempt of cryptanalysis of the HFE system leads us to study boolean functions and their representation within BDD (Binary Decision Diagram) framework. Some results in the complexity of representing some boolean functions have been obtained.

We based our investigation on truth tables, but taking hard graphs and interpreting them with different semantics (euclidian division of polynomials for example) may lead to different sets of hard functions, studying them will certainly be interesting.

Chapter 7

Conclusion

The most important part of this work begin by studying the algorithmic behavior of some imperative programming languages and leads to a questioning on the formal definition of the concept of algorithm.

The list of algorithmic shortcomings of programming languages is not enough and a more constructive process is necessary. For this, I relied on the real progress made in defining the formal notion of algorithm (especially the one of Y. Gurevich with abstract state machines, ASM) to arrive at a definition of a class of algorithms (one to calculate the primitive recursive functions).

My work has ignored the formalization of the algorithms made by Y. Moschovakis even if the search for functional systems that are algorithmically complete for the class of primitive recursive algorithms is a partial answer. This research is an important motivation for me. This is the direction I want to take in the coming years.

To my opinion, ASMs have yet generated what they deserved and work on evolving multi-algebra is an illustration. The reformulation of the models of computation in a unified form is an example of the expressive power of evolving algebras. This is the second topic I wish to continue in the coming years, especially extending the representation of computational models to programming languages.

The part on boolean functions and binary decision diagram has been a great experience, for me, in the field of cryptography even if this field is a pretext to study theoretical concepts. I do not despair to pursue the study of representation of special boolean functions with other tools coming from computer science.

Bibliography

[1] E.R. Berlekamp. *Factoring Polynomials Over Finite Fields.* Bell Systems Technical Journal, Vol. 46, pp. 1853–1859. 1967.

[2] R.E. Bryant. *Graph-based algorithms for boolean functions.* In IEEE Transactions on Computers. C-35(8), pp. 677–691. 1986.

[3] N. Courtois. *The Security of Hidden Filed Equations (HFE).* `http://www.cryptosystem.net/hfe/`.

[4] J.C. Faugère, A. Joux. *Algebraic cryptanalysis of Hidden Field Equation (HFE) cryptosystems using Gröbner bases.* In proceedings, Advances in Cryptology - CRYPTO 2003, D. Boneh eds, Lecture Notes in Computer Science, LNCS 2729, pp. 44-60. 2003.

[5] E. Férard, F. Rodier. *Nonlinearity of some Boolean functions.* In proceedings, BFCA'06, J-F. Michon, P. Valarcher, J-B. Yunes eds, Publications des Universits de Rouen et du Havre.2006.

[6] M.R. Garey, D.S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness.* W.H. Freeman Company. 1979.

[7] L. Granboulan, A. Joux, J. Stern. *Inverting HFE is Quasipolynomial.* In proceedings, CRYPTO'2006, Lecture Notes in Computer Science, LNCS 4117, pp. 345–356. 2006.

[8] C. Gröpl. *Binary Decision Diagrams for Random Boolean Functions.* PhD Thesis, Humboldt-Universitt zu Berlin. 1999.

[9] C. Gröpl, H.J. Prömel, A. Srivastav. *Size and structure of random ordered binary decision diagrams (extended abstract).* In proceedings, STACS'98, Lecture Notes on Computer Science, LNCS 1373, pp. 238–248. 1998.

[10] C. Gröpl, H.J. Prömel, A. Srivastav. *On the evolution of the worst-case obdd size.* Information Processing Letters, Vol. 77, pp. 1–7. 2001.

[11] X. Jiang, J. Ding, L. Hu. *Kipnis-Shamir's Attack on HFE Revisited.* 3rd International SKLOIS Conference on Information Security and Cryptology, Inscrypt 2007 (formerly CISC). August-September 2007, Xining, China. 2007. Lectures Notes in Computer Science, LNCS. 2007.

[12] A. Kipnis, A. Shamir. *Cryptanalysis of the HFE public key cryptosystem by relinearization.* In proceedings, CRYPTO'99. Lectures Notes in Computer Science, LNCS 1666, pp. 19–30. 1999.

[13] D.E. Knuth. *The Art of Computer Programming.* Vol. 2. Addison-Wesley. 1998.

[14] T. Matsumoto, H. Imai. *Public quadratic polynomial-tuples for efficient signature-verification and message encryption.* In proceedings, Advances in Cryptology EUROCRYPT'88, LNCS 330, pp. 419–453. 1988.

[15] J.-F. Michon, P. Valarcher, J.-B. Yunès. *Sequence of Enumeration of QROBDD.* On-Line Encyclopedia of Integer Sequences. Sequence A100344. 2004. `http://www.research.att.com/ njas/sequences/`

[16] J.-F. Michon, P. Valarcher, J.-B. Yunès. *On Maximal QROBDD's of Boolean Functions.* RAIRO ITA/TIA, Vol. 39(4), October-December 2005. Ref. ITA0442.

[17] J.-F. Michon, P. Valarcher, J.-B. Yunès. *HFE and BDDs: A Practical Attempt at Cryptanalysis.* In Proceedings, International Workshop on Coding, Cryptography and Combinatorics, CCC'03, China. K. Q. Feng, H. Niederreiter, C. P. Xing, eds. Progress in Computer Science and Applied Logic, Vol. 23, pp. 237–246. Birkhaser, 2004. isbn 3-7643-2429-5.

[18] J.-F. Michon, J.-B. Yunès. *A HFE cryptosystem generator.* `http://www.liafa.jussieu.fr/~yunes/HFE/`

[19] S.I. Minato, N. Ishiura, S. Yajima. *Shared binary decision diagram with attributed edges for efficient boolean function manipulation.* In proceedings, 27th ACM/IEEE Design Automaton Conference, Orlando, FL, pp. 52–57. 1991.

[20] *NTL Library.* `http://www.shoup.net/`

[21] J. Patarin. *Cryptanalysis of the Matsumoto and Imai public key scheme of Eurocrypt'88.* In proceedings, Advances in Cryptology - CRYPTO'95. Lecture Notes in Computer Science, LNCS 963, pp. 248–261. 1995.

[22] J. Patarin. *Hidden Fields equations (HFE) and isomorphim of polynomials (IP): two new families of asymetric algorithms.* In proceedings, EUROCRYPT'96. Lecture Notes in Computer Science, LNCS 1070, pp. 33–48. 1996.

[23] I. Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications* SIAM Monographs on Discrete Mathematics and Applications. 2000.

[24] I. Wegener. *The Complexity of Boolean Functions.* John Wiley and Sons. 1987.

[25] J. von zur Gathen, J. Gerhard. *Modern Computer Algebra. 2nd edition.* Cambridge University Press. 2003.

[26] M.J. Williamson. *Non-secret Encryption Using Finite Field.* 1974.

[27] J.-B. Yunès. *HFE experiments.* `http://www.liafa.jussieu.fr/~yunes/HFE/`

[28] J.-B. Yunès. *A ROBDD package.* `http://www.liafa.jussieu.fr/~yunes/HFE/`