PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 42, NO. 1, PP. 5-24 (1998)

TOWARDS MODELS OF REALISTIC COMPUTING MACHINES IN COMPUTER SCIENCE

Monica ALDERIGHI*, Andrea BORDONI**, Giovani A. MOJOLI**, Alessandro SALA***, Giacomo R. SECHI* and S. VINATI**

* Istituto di Fisica Cosmica e Tecnologie Relative
Consiglio Nazionale delle Ricerche
Via Bassini 15, I-20133 Milano, Italy
Tel: +39-2-23699332, Fax: +39-2-2362946,
Email:{monica,giacomo}@ifctr.mi.cnr.it
** Dipartimento di Fisica
Universita' degli Studi di Milano
Via Celoria 21, I-20133 Milano, Italy
*** Dipartimento di Matematica
Universita' degli Studi di Milano
Via Saldini 50, I-20133 Milano, Italy

Received: September 30, 1997

Abstract

The paper presents an approach to system modelling in design of both hardware and software systems. It is based on the definition of models of machines that can be directly implemented. The paper shows how to render less abstract and more realistic the abstract machines defined by theoreticians, so that they can capture implementation and technological-oriented aspects, such as testability, and allow an easy transition to final implementations. A realistic abstract machine for lambda-calculus is then presented and the design of system for lambda-expressions evaluation is illustrated. The architecture chosen for the system is based on a collection of finite state automata, evolving concurrently and communicating via a broadcast system. Some conclusive remarks about the use of realistic models are finally drawn.

Keywords: system models abstract machines, system design, functional languages implementation.

1. Introduction

The central issue of computing system design is the definition of models. Models are conceptual descriptions of systems, highlighting their most significant features and ignoring others, immaterial for the considered level of abstraction. Design can be viewed as a process passing through different levels of descriptions from requirement analysis to final implementations. Regardless the specific design methodologies adopted, hardware and software design aims at filling the gap between two worlds, one highly abstract, in which problems are formulated, the other more concrete, in which the solutions to the problems are implemented by target technologies.

The validation and verification of actual implementations are the ultimate goals of design and also here models play a fundamental role. On the basis of the agreement between the implementations and their specifications (models), the implementations goodness can be assessed. Verification involves checking that the built system, a program or an electronic device, conforms to its specification, while validation involves checking that the built system as implemented meets the expectations of the user.

Testing is an important part of validation and verification of implementations, as these cannot be assumed to be error-free. Testing has been the only post-implementation validation and verification technique for a long time. Testing is a dynamic technique. It involves exercising the implementation on a set of suitable test cases, observing its behaviour and looking for unexpected behaviours. Nowadays testing is used together with static techniques, such as system inspections, analysis and formal verifications. Although the presence of static techniques, the role of testing remains fundamental, as static technique can only check the correspondence between an implementation and its specification, and cannot reveal non-functional characteristics of an implementation.

Testing is to be understood as a mean to reveal the presence of errors in implementations. It can never show its correctness and the absence of errors. In this sense, a successful test can be considered one which establishes the presence of one or more errors in the built system. The emphasis given to design for testability techniques witnesses the importance of testing, as a mean for observing and measure the behaviour of implemented systems.

The paper presents an approach that contributes to system modelling in design. The approach deals with the design of machines, but it can be extended to design of programs as well. We are interested in the definition of models of machines that can be directly implemented, and subject of the paper is one of the possible ways to achieve this. It is shown how to render less abstract and more realistic the abstract machines defined by theoreticians, so that they can capture implementation and technological-oriented aspects, such as testability, and allow an easy transition to implementation.

Specifically we focus on lambda-machines as they constitute a very interesting subject of study. Indeed lambda-calculus is the representative of functional languages and in general of all context sensitive languages, which are very important for the definition of new computational paradigms (BACKUS, 1978). Moreover lambda-calculus is adopted in order to give theoretical foundations of computation (GIRARD et al., 1989). For instance in intuitionistic type theory lambda-calculus is used as theory of expression and the reduction rules are defined as inference rules of the theory (NORDSTRÖM et al., 1990).

Aim of the paper is the definition of a realistic abstract machine for lambda-calculus and the design of a system for lambda-expression evaluation. The architecture chosen is based on collection of finite state automata, evolving concurrently and communicating via a broadcast system.

At first glance the design and implementation of a machine for lambda-calculus may appear a mere academic exercise, a completely unuseful task from a practical point of view. We think, instead, this study can be useful to investigate and understand more general subjects, such as how to develop high-level machines that employ alternative computational paradigms and languages, and how technology features and specific design requisites, such as testability and error detection, impact on the model of the system to be built and on the formal definition of the language.

Section 2 overviews the main contributions to lambda-calculus and related topics. The characteristics of digital system design are summarised in Section 3. Section 4 presents an analysis of lambda-machines, as found in literature and introduces the requisites for realistic machine models. A suitable definition of lambda-calculus is given in Section 5, while the specifications of data for representing the corresponding lambda-expressions are illustrated in Section 6. A realistic lambda-machine model is then shown in Section 7, and finally the architecture of the a system implementing the realistic model devised is sketched in Section 8.

2. Related Works

The functional approach as response to the need of being liberated from the von Neumann style, both in programming and in computer machines is firstly presented by Backus in (BACKUS, 1978). He claims for machines showing greater mathematical properties, in order to prove facts about programs. He also intends to improve the foundation of denotational semantics, because denotational semantics makes it possible to understand the domain and spaces of functions, implicitly involved in the traditional programs and computational models, such as Turing machine, finite state automata, etc. At the same time with FP and FFP languages, he provides a theoretical formulation of the operational semantics to its functional machines.

The fundamental dichotomy in logic between syntax, proofs and sense on one hand, and denotation, semantics and algebra on the other is investigated by Girard in (GIRARD et al., 1989). In an attempt to reduce this gap, he explains that the tradition of denotational semantics is more developed than that of syntax or operational semantics. Operational semantics is very important in computer science, for it is the common foundation for defining both abstract and real machines. However, computer science posed very important and partially unsolved theoretical problems, that caused, at least partially, a disaster for an evolution of operational semantics parallel to that of denotational semantics.

A basic theoretical support to operational semantics, putting in evidence the underlying logic and its symmetries, can be offered by the logistic calculus of Gentzen (GENTZEN, 1969).

On this basis, Hannan (HANNAN et al., 1990, HANNAN, 1991) connects demonstration theory and sequent calculus with abstract machines, providing thus an operational semantics to the abstract machines. The abstract machines are presented by theoreticians as frameworks for studying the operational semantics of programs; they are often introduced an intermediate level for the machine description in order to implement the programming languages. Hannan generalises the concept and presents systems of abstract evaluations produced by means of correctness preserving transformations.

The prototype or archetype of abstract functional machine is the Stack, Environment, Control and Dump machine (SECD) by Landin (LANDIN, 1964), whose correctness proof is provided by Plotkin (PLOTKIN, 1975). Successive developments are mainly improvements of this machine, employing more accurate models of functional languages, such as category theory, instead of Landin's applicative structures.

The syntactic equivalence between lambda-calculus and category theory is illustrated by Cousineau in (COUSINEAU et al., 1987). According to this equivalence, a Categorical Abstract Machine (CAM) is shown. It processes abstractions similarly to the SECD and makes it possible a simpler proof of the semantics correctness.

One of the peculiar characteristics of the CAM and other abstract machines, such as the reduction machine by Turner (TURNER, 1979), is that they need a compiler for translating the programs (lambda-expressions or functional programs) into proper strings of machine instructions. Therefore the correctness proof of the machine programs depends on both the compiler and the machine.

Other interesting examples of abstract machines are the Three Instructions Machine (TIM) (FAIRBAIRN et al., 1987) and the Linear Abstract Machine (LAM) (LAFONT, 1988).

3. On Digital System Design

Design deals with models at different levels of abstractions. Also at the lowest level, it deals with models of technology rather than with a specific technology, either software or hardware. In order to satisfy requisites of testability and error detection in implementation, models are to have some characteristics. For instance, in the case of digital system design considered in this paper, they possess the following features:

- They are discrete systems, i.e. entities whose behaviour and functioning are described by discrete quantities, in time and resources;
- They are structured in parts, sub-systems;
- Sub-systems can be analysed independently from each other. A subsystem can be modified or replaced without altering the remaining sub-

systems. This is a necessary condition to avoid that errors occurring in a sub-system affect other sub-systems;

- Sub-systems should be directly implemented in the chosen target technology;
- Sub-system implementations can be tested independently from each others. This is a necessary precondition for design testability.

Systems communicate through defined input/output channels. These are the only means to access the internal environment of a system. Also sub-systems have their own environment, contained in that of the system of which they are part of, and that can be accessed only through input/output channels.

Communication among sub-systems is allowed by means of connection links. The architecture of a system defines how the system is structured in sub-systems and how these communicates.

Testing of systems requires establishing the environmental conditions for carrying out the test experiment, the quantities to be measured and the measurement apparatus. Locality of system's environment cannot be violated during evaluation.

Decomposition of system in sub-systems improves testability allowing to test separately each sub-system first, and then the entire system. Both at system and sub-system level, the test set consists of the Cartesian product of all the possible sequences of input data.

In case of unexpected behaviours, the system must provide information making it possible to diagnose what happened.

4. Abstract Lambda-Machines

4.1. Defined Lambda-Machines

In literature theoreticians define abstract lambda-machines as automatic interpreters of lambda-expressions. Implementations are written in very high level languages, such as ML (INRIA, 1984) and ALF (MAGNUSSON et al., 1994). These languages are based on assumptions that are theoretical extensions of lambda-calculus and the implementations are, thus, very clean and understandable. However, they are not suited to our purpose, that is to design and implement a machine performing a formally specified calculus. An attempt to implement either mentioned machines, by adopting a very simple hardware technology (modelled by discrete systems), shows critical aspects with respect to testability and correctness issues.

From a design point of view, the definitions of the abstract machines analysed (e.g. SECD, CAM and TIM) are equivalent. Indeed they are relatively simple and are completely specified in terms of a set of instructions,

a computational environment, and the operational semantics of each instruction, showing the effects of instruction execution on the computational environment.

For example, let us consider the CAM. Instructions are listed in *Table 1* together with associated operational semantics.

Table 1. CAM instructions definition

Pre-execution			Post-execution		
Term	Code	Stack	Term	Code	Stack
(s,t)	Fst;C	S	s	C	S
(s,t)	Snd;C	S	t	C	S
s	Quote(c);C	S	c	C	S
s	Cur; (C)C1	S	(C:s)	C1	S
s	Push; C	S	s	C	s.S
t	Swap; C	s.S	s	C	t.S
t	Cons; C	s.S	(s,t)	C	S
(C:s,t)	App; C1	S	(s,t)	C; C1	S

A model of the CAM described as a discrete system would consist of: three sub-systems corresponding to three data-structures term, code and stack; some sub-systems, in charge of performing the eight instructions; a control sub-system; connections among sub-systems.

The critical aspects are basically due to two questions. First, the technological implementation adds details, other than those theoretically specified and whose definition needs ingenuity. For instance, the two data structures term and code are considered as simple stacks. However, they are more complex than stacks, because each element is a string of varying length with a marker as terminator. If stacks are well known structures, formally well defined and easy to implement, stacks of lists, as term and code are, are functionally comparable to trees and need to be handled by recursive mechanisms, not included in the theoretical specifications.

The second question is that at the system level the machine has to be tested on sequences of lambda-expressions. But, as the machine interprets the instruction strings produced by the compiler, testing needs to be carried out on the outcomes of the compiling function.

Error detection, thus, requires detecting possible compiler errors, especially because the machine interprets a domain of instruction sequences larger than the range of the compiling function. If we specify a correct operation as the achievement of a finale state starting from an initial one, for instance a final state with term full, and code and stack empty from an initial state, consisting of a code full of instructions, then there exists an arbitrary number of sequences of instructions, satisfying the requisites of

correct operation that are not lambda-expressions.

Let us consider the following initial state:

Term	Code	Stack
[]	{((),a),b) push;swap;cons []}	[]

After applying the necessary rules, the machine is in the following state:

Term	Code	Stack
{ ((),a),b),nil) [] []}.	[]	[]

The behaviour of the machine agrees with specifications at the level of single instructions, but not at the level of sequences of instructions. The sequence presented does not correspond to any lambda-term. The previous situation might be a symptom of a compiler error, such as the partial construction of a lambda-term.

We are going to introduce new concepts in the abstract specification of lambda-calculus that make it possible to define a realistic abstract machine. We intend to build a simple system, whose functioning is described and foreseeable according to the abstract model and can be tested in a limited amount of time.

4.2. Realistic Machine Requisites

The design of a realistic machine implementing a symbolic calculus requires both a suitable specification of the calculus and a description of the machine. The dynamic evolution of the machine is to correspond to the operational semantics of the calculus, formally specified.

Assuming a model of real technology as given, design is now to face with how to build a realistic description of the machine, and how far this description is from the final implementation, or how well the implementation is described by the chosen model.

Establishing correspondences between a real machine and the formal specification (theory) of the calculus it performs is impossible, due to the intrinsic infiniteness of theories. From now on, we will use expressions as 'a real system is modelled by a theory' or 'the description (model) of a real system is consistent with a theory' in a weaker meaning. We mean that machine outcomes are consistent with the predictions of the theory on the finite subset of objects, on which the machine operations are defined. If such subset were not closed with respect to the operations, it would be necessary to 'close' it, possibly re-defining it or the operatins themselves. With this restriction the validity of theory predicates may not be preserved and, then, exploited.

Mathematical infinity is a concept unrelated to technological implementation. Mathematical finiteness can be quite unrelated as well, although

it involves an arbitrary limited number of objects. Indeed the set can be so large that no real machine is able to handle it.

Real machines deal with a stricter concept of finiteness. This refers to having an upper bounded number of elements that can be treated by means of enumeration. This number has to be compatible with the chosen technology.

Our approach follows two directions, one is technology- and the other is theory-oriented. As far as theories are concerned, we try to reduce the need of induction and infinity in the multiplicity of the objects considered, by adopting enumerative sets as basic ingredients. As far as technology is concerned, we propose *realistic models* of machine.

The basis for the definition of a realistic machine model is the following list of requirements. They represent a first step towards collecting the common features of testable systems, modelled as discrete systems. In this way they are not meant as definitive or exhaustive. At the higher level, theories have to be consistent with these requisites and every time a specific feature has to be included in the realistic model, both the realistic model and the corresponding theoretical apparatus are reviewed, in order to maintain the consistency between them. The following requirements have been defined:

- Δ-finiteness in area: computing resources must have fixed upper-bounds;
- Δ-finiteness in time: execution time must be finite and fixed; this and the previous constitute the minimal requisites granting system construction.
- falsifiability: this is relevant to testability purposes.

We define a system as falsifiable if its model is falsifiable (POPPER, 1970). If the theory admits a realistic logical model and this is consistent, then this requirement coincides with testability.

In case it is necessary to verify this feature by means of a system described by the same quantities, a stronger formulation is required. A system is then said Ω-falsifiable, if it is falsifiable by means of a system of lower complexity. By complexity we mean here a measure of some properties of the system, significant for the level of abstraction concerned.

Although weaker requirements not including falsifiability can be employed to build systems, they result in a loss of testability.

Lambda-calculus is well suited to our purposes, for it is theoretical enough to be used in higher order theories and is described by means of a limited and fixed set of rules.

5. Some Remarks on Lambda-Calculus

As previously mentioned we have chosen lambda-calculus to illustrate the impact of realistic assumptions on the development of a real machine imple-

menting formally specified computations. In particular we chose $\alpha\beta$-untyped lambda-calculus because it has a high expressive power, nevertheless its formulation is rather simple. The evaluation of its expressions is generally context dependent, and this can be seen as the most abstract and general way of handling arbitrary complex expressions, yet only few definitions are necessary to describe it completely.

The objective is the definition of a machine modelling manipulations of lambda-terms by application of α and β formal rules. The machine is to accomplish the automatic reduction of lambda-terms to possible normal forms or some generic transformations of lambda-terms via β-reductions.

Let us report two among the possible formulations of lambda-calculus, respectively by Barendregt (BARENDREGT, 1984) and Revesz (REVESZ, 1988). In both formulations, terms are built from an infinite alphabet $v_1 \ldots v_n \ldots$ of variables, and two generators, Λ, which is the abstraction and ( ), which is the application.

The set Λ of lambda-terms is inductively defined by a finite number of rules:

$x \in L$;
$M \in \Lambda \Rightarrow (\lambda x.M) \in \Lambda$;
$M, N \in \Lambda \Rightarrow MN \in \Lambda$ (or $(M)N \in \Lambda$);

where x represents an arbitrary variable.

We have adopted a Gentzen like style of notation for term definition (ASPERTI, 1991). This is well suited to express the bottom-up constructive approach used in the proof of a lambda-term, once its sub-terms are proved. The machine adopts this kind of rules to constructively prove lambda-terms to be well formed. This avoids the use of recursive application of inductive terms definitions. The resulting rules for term definition are listed here below:

$\frac{x \in \text{var}}{x \in \Lambda}$ var introduction

$\frac{x \in \text{var} \wedge M \in \Lambda}{\lambda x.M \in \Lambda}$ λ introduction

$\frac{x \in \Lambda \wedge M \in \Lambda}{(N)M \in \Lambda}$ app. introduction

$\wedge$ and $\in$ connectives belong to the proof's meta language.

An occurrence of variable v_n is said free if it is not in the scope of a $\lambda.v_n.$, it is said bound otherwise.

The introduction of conversion rules gives new quotient sets of the set Λ. The α and β rules described according to Barendregt notation are as follows:

α-rule:
Two terms M and N are α-congruent iff M is a change of bound variables of N.
Given $M_1 \ldots M_n$ α-congruent terms, if they occur in a certain context then all bound variables are declared different.

β-rule:

- $(\lambda x.M)N = M[x := N]$, ($\beta$-conversion)

Axiomatic properties of equality:

- $M = N \Rightarrow MZ = NZ$;
- $M = N \Rightarrow ZM = ZN$;
- $M = N \Rightarrow \lambda x.M = \lambda x.N$;

Substitution rule:

- $x[x := N] \equiv N$;
- $y[x := N] \equiv y$, if $x \neq y$;
- $(\lambda y.M_1)[x := N] \equiv \lambda y.(M_1[x := N])$;
- $(M_1 M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N])$;

The α and β rules described according to Revesz notation are instead:

α-rule:

Def: renaming of a variable.

- $\{z/x\}x \equiv z$;
- $\{z/x\}y \equiv y$; if $x \neq y$;
- $\{z/x\}\lambda x.E \equiv \lambda z.\{z/x\}E \ \forall E \in \Lambda$;
- $\{z/x\}\lambda y.E \equiv \lambda y.\{z/x\}E \ \forall E \in \Lambda$, if $x \neq y$;
- $\{z/x\}(E_1)E_2 \equiv \{\{z/x\}E_1)\{z/x\}E_2 \ \forall \ E_1E_2 \in \Lambda$.

α-rule: $\lambda x.E \rightarrow \lambda z.\{z/x\}E \ \forall z$ which is neither free nor bound in E.

β-rule: $(\lambda x.P)Q \rightarrow [Q/x]P$

note: α and β rules are equivalence relations.

β-rule:

Def: the substitution of a term.

- $[Q/x]x \cong Q$;
- $[Q/x]y \cong y$; if $x \neq y$;
- $[Q/x]\lambda x.E \cong \lambda x.E \ \forall E \in \Lambda$;
- $[Q/x]\lambda y.E \cong \lambda y.[Q/x]E \ \forall E \in \Lambda$, if $x \neq y$ and at least one condition holds:
- $x \notin \text{Free}(E)$, $y \notin \text{Free}(Q)$;
- $[Q/x]\lambda y.E \cong \lambda z.[Q/x]\{z/y\}E \ \forall E \in \Lambda$ and $\forall z$ with $x \neq z \neq y$ which is neither free nor bound in $E(Q)$, if $x \neq y$ and both $x \in \text{Free}(E)$ and $y \in$Free(Q) hold;
- $(E_1)E_2 \cong ([Q/x]E_1)[Q/x]E_2$

In this paper Barendregt's formulation has been adopted, as it requires that the meaning of variables is univocally defined before applying β conversions. In Revesz's formulation, checks on the context dependency of variables are performed as evaluation is taking place.

Fig. 1 shows three possible representations of the same lambda-term. Even though equivalent with respect to the theory, they underline different aspects. Strings highlight the fine structuring of terms as successions of

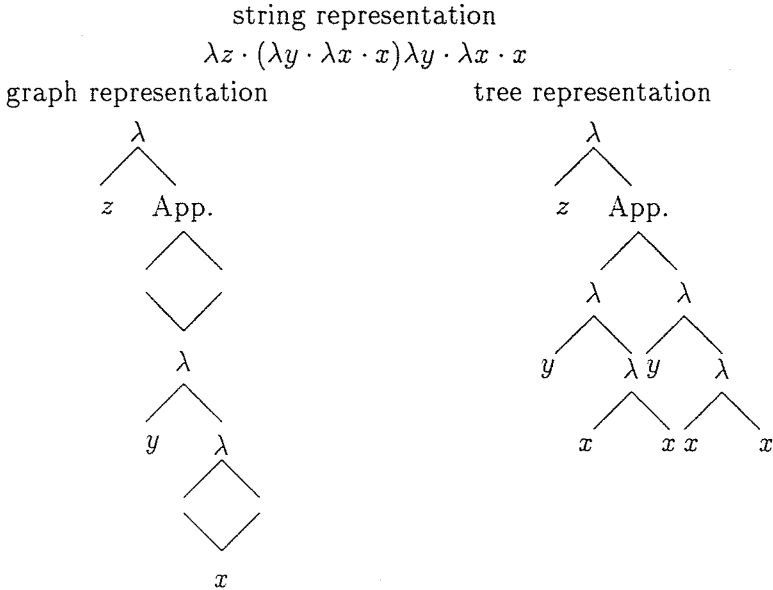

Fig. 1. Equivalent representations of lambda-terms

symbols. This characterisation is so simple and general to make it possible to implement terms and transformations on them on any conventional sequential machine. From a theoretical point of view, strings are the most suitable representation for a Turing machine. Yet this representation does not support direct visibility of sub-terms, i.e. a sub-string may not be a sub-term. Graph representations permit uniform treatments: each sub-term is a graph itself. Graphs, by adopting referential transparency on terms, allow to implement context free operations on (sub-)terms efficiently. Trees, instead, allow to distinguish among different occurrences of the same (sub-)term, and are thus well suited to context dependent evaluations.

6. Data Specifications

Implementing lambda-terms transformations according to lambda-calculus operation semantics requires a richer description than those usually carried out by strings. What we are claiming here is that this requires recognizing the syntactical-semantic nature embedded in the terms, e.g. that they are made by sub-terms. Intuitively speaking, we want the machine to view a lambda-expression not as a pure sequence of symbols, but with the same 'depth' theoreticians have when treating them. Graphs are adopted as inter-

nal machine representation. A string notation is used for external communications in order to provide easy interface to the user. Automatic translation from and to the user is performed by the machine.

For external representation, the following definition holds.

- lambda-term: **k-finite** string of characters from the alphabet, consisting of:
  - $a, b, \ldots, z$ variables identifiers
  - $\lambda.$ abstraction symbols
  - () application symbols
- $(A)B$: application of A to B (association to the left)
- $\lambda c \cdot B$: abstraction of B on c given A, B, c, lambda-terms, and c variable.

No conventions on how distinguishing free and bound occurrences of variables are required.

In internal representation, lambda-terms are represented and managed inside the machine as binary acyclic directed graphs: vertexes correspond to abstraction and application, leaves correspond to atomic terms (variables). Each vertex is root of its sub-term graph, which is thus univocally denoted. The vertex is viewed as abstraction of the sub-term and contains the information of its outermost constructor.

7. Functionalities Specifications

Given the specified data representations used for input/output communication and computation, we now proceed to the definition of a high-level architecture via functional description of machine tasks. Realistic requirements suggest to maintain different tasks, mapping them into different and distinguished sub-machines.

During processing, the machine undergoes the following activities:

1. Acquisition of a string and construction of its internal graph representation;
2. Evaluation of lambda-term via α and β conversions;
3. Output production by means of graph-to-string conversions.

In order to accomplish these activities two different data areas are used, named static and dynamic. Their characteristics are highlighted in the following Sections.

7.1. Static and Dynamic Areas

Two kinds of operations on terms can be distinguished: the former, called static or conservative, preserve terms arguments possibly making them part of a supra-term; the latter, called dynamic, transform terms arguments into new different terms. Examples of conservative operators are bottom-up term constructors, while examples of dynamic operations are α and β reductions. Occurrences of the same sub-term may have different meanings in the dynamic world, while conservative operators can manage them as a single term and are therefore compatible with referential transparency.

Static and dynamic operations are defined in the static and dynamic area respectively. In the static area, data (terms) are represented by graphs, while trees are used for representing data in the dynamic area.

7.2. Strings Acquisition

A string is read from left to right and the corresponding binary tree is built top-down into the dynamic area. This operation also verifies string syntax. If the procedure ends (and ends correctly) in a number k of steps, this means the input string is syntactically correct and the tree represents it. A positive outcome of the syntactic check proves that the string is an effective representation of a lambda-term and the existence of a constructive proof of the term itself by j applications of Gentzen-like rules, with j depending on k only.

The syntactic check does not provide this proof, because during the process intermediate structures are created that are not properly interpreted as terms. The machine has to prove constructively the terms by means of application of the bottom-up rules peculiar to the set Λ, in order to handle them.

Indeed syntactic check operators are defined on strings and return values belonging to a subset larger than the tree terms set, while constructors are onto the set Λ.

Given a tree in the dynamic area, the corresponding graph term is then built in the static area. The process starts from the leaves of the tree and proceeds upwards until a root is reached. For every node encountered in the traversal, a corresponding vertex is put in the graph. This *de facto* constitutes the proof of the lambda-term. Graph vertexes are operators defined according to the three construction inference rules presented in Section 5. The application of an operator to a term creates a new term. The order of application of this operator is dictated by the bottom-up traversal of the tree in the dynamic area. The following correspondences hold:

dynamic area	**static area**
variable node	atom operator
application node	application operator
abstraction node	abstraction operator.

Static area always contains well formed terms, unless some static operator has been defined incorrectly, or some errors have occurred; this last aspect falls beyond the scope of the paper. Possible erroneous applications of operators in the dynamic area yield unexpected, yet well formed terms without affecting static area. Creation and proof of a term implies creation and proof of all its sub-terms. During all intermediate steps of the process static area contains sub-terms as well formed terms. There is no difference here between term and sub-terms. As a matter of fact static area is deputed to context free operations.

Once the static term is built, the representation in the dynamic area is now redundant and thus is removed. Further operations on terms will construct the dynamic context dependent representation from the static term, will manage it and then build the static image of the obtained term.

The separation between static and dynamic areas obeys to the realistic constraints of testability.

7.3. Terms Evaluations

By means of α conversion the bound occurrences of every variable are renamed by unused name. According to Barendregt (BARENDREGT, 1984) this operation is preliminary to every β reduction step. For the realistic requisite, Barendregt's definition of β reduction via substitutions is adopted having a more convenient automatization with respect to usual definitions (REVESZ, 1988; HINDLEY, 1986). This requires terms to be written according to Barendregt renaming convention, otherwise incorrect interpretations of variables roles may occur; new names label bound variables, unused for free variables. We choose a stricter convention adopting different names for different bounds, formally identifying the scope of each lambda with their labels.

β conversion is obtained in four steps.

1. The dynamic context dependent representation is built from the static term.
2. The dynamic term obtained is α converted.
3. A $1-\beta$ restricted reduction (defined only on a converted terms) is made.
4. The static image of the resulting term is built.

$1-\beta$ restricted reduction is the transformation of a term via β-rule onto a single redex occurrence in the tree. This choice better agrees with realistic requirements. Indeed there is no simple way to assure that a generic reduction will not exceed machine k-bound resources and admissible j-limited

execution time. Time and resources necessary for $1-\beta$ application, instead, can be foreseen by the actual term, independently from terms previously obtained by means of β-conversions. Algorithmic complexity decreases if the contextual dependency is reduced. Reduction generally involves the application of a given strategy to the whole term and requires a heavy additional work to grant a limited and prefixed execution time. Unfortunately the complexity of this work is comparable with that of reduction itself. Moreover, the correctness proof of a generic β reduction algorithm would require to evaluate it with respect to all the reduction strategies on all the possible terms. Instead, the correctness evaluation of $1-\beta$ reduction is strategy independent.

A generic β reduction may be split in some $1-\beta$ steps; our choice thus involves no limitations of model expressiveness and supports every reduction strategy.

In order to ensure the correctness of the entire process, the correctness of β operator has to be proved. The problem may be simplified by distinguishing among the proof that β operator is well defined on its domain and the proof that it applies only to data belonging to its domain. The result of a $1-\beta$ reduction is not a proved well-formed term; then further direct re-applications of the β operator may therefore be undefined and yield unpredictable results.

The static image of the term obtained by $1-\beta$ operator is built in step 4. This proves the result is a well-formed term, thus preventing successive application of $1-\beta$ operator to incorrect data, and partially checks $1-\beta$ operators correctness. All intermediate terms obtained by $1-\beta$ steps are present in the static area. They provide successive snapshots of the term evolution and allow simpler proof of the correct execution.

7.4. Output Production

At this step the static representation of a term is converted into string representation. It is worthwhile reminding the only proved well-formed terms in the machine are static terms.

8. From Realistic Towards Real Machines

According to realistic guidelines, operations required by $\alpha\beta$-untyped lambda-calculus have been partitioned, in order to reduce algorithmic complexity without loss of expressiveness. Activities have been realised by means of a limited number of independent modules whose execution is bound in resources and time. Operators have been distinguished on the basis of their context dependency, and different representations have been chosen for their corresponding domains.

Realistic constraints defined in Section 4.2 (Δ-finiteness and falsifiability) are expressed at a higher abstraction level than that of the target implementation. Due to the lack of any other constraint (e.g. real time constraints), the realistic machine sketched above is rather independent from technological and architectural choices and, therefore, it models a class of real machines.

In order to identify and define a particular member of the class, some real constraints have to be fixed. The introduction of real constraints in the system description may cause mismatching between the resulting system and the realistic model of the machine. These constraints may even impact on the rules of the formal calculus, possibly yielding different or restricted definition of its characteristics. Realistic requirements may be partially disregarded, depending on the specifications and on the chosen technology (e.g. when the correct execution of the real-system cannot be proved).

In the following Sections, the design of a system for lambda-expressions evaluation will be briefly illustrated. A detailed, application independent, description of the chosen architecture is out of the scope of this work. Interested readers may refer to (BORDONI et al., 1997).

8.1. System Architecture

The system consists of three different sub-systems (modules). Each module is viewed as a distributed system and described in terms of a collection of communicating finite-state automata (ALDERIGHI et al., 1997a), called broadcast automata. The following modules are identified: interface, static and dynamic units.

The interface unit is to manage commands acquisition, user addressed information (e.g. state information) and data exchange (i.e. mapping between internal and external representation).

The static unit is to perform constructive proof of acquired terms, and serves as static typed memory of graph terms with referential transparency. Dynamic context dependent tree terms have to be built up from data stored in this unit. It, thus, requires a database structure containing information about existing terms and it also has to map graph internal representation into strings.

The dynamic unit is to perform the syntactic check of the acquired strings and the execution of all the structural manipulations defined on lambda-terms.

The sharp separation among these units agrees with the realistic model of the machine. However, functioning of the machine requires heavy and continuous interactions among these units. These interactions are to be carefully modelled.

Similarity between tree and graph representations suggests implement-

ing static and dynamic units by means of homogeneous automata. Static and dynamic units are then defined in terms of two different *families* of homogeneous automata.

The automata in a given family are described by the same finite state machine model, with the exception of the initial state. Each automaton knows the state of every automaton of the family. Direct communication between non homogeneous automata is not allowed. Each family describes, from a geometrical point of view, a complete graph, whose vertexes are its automata and edges are the connections among them.

Term representations are obtained by means of stable *affinity* relationships among automata (ALDERIGHI, 1997b). An automaton A is said affine to automaton B if the actual evaluation of the transition function of A is not degenerate on the state value of B. The relationship is k-stable if it maintained for a prefixed number k of steps.

Communication among different families is allowed by means of a specialised automaton. The transition function of an automaton depends on its current state, the current state of the other automata in the family, and on information coming from other automata families. Each transition updates the automaton state and produces information for non homogeneous automata.

Broadcast automata exploit a control activity which is spread over all automata. An operation is the result of the combined effect of single automata evaluations. So algorithms have to be split onto single automata transition rules. This architecture, while comporting a loss of efficiency with respect to centralised architectures, enhances concurrence and allows abstract management of (sub-)terms. If an operator handles a sub-set of (sub-)terms in the same way, it is possible to re-define it by abstraction on the particular representative of the proper class of equivalence.

Term representations are embedded in the graph structures of the corresponding families. Each vertex of the representation is associated to a vertex of the family graph. This allows implementation of abstract evaluations and disregards fine characteristics of sub-terms by means of local transitions, in the meaning of abstracting a sub-term by a node as explained at the end of Section 6.

Unfortunately, some algorithms (e.g. algorithms implementing con text-dependent operators, such as α conversion) may require distributed information everywhere in the tree, even though the information is split on a few automata. Therefore the transition rules of an automaton have to be defined on actual state of all family automata, independently of established affinities.

8.2. Dynamic Family

A tree is obtained by means of affinities established among automata in the complete graph of the family. To each automaton, or vertex of the family graph, corresponds a node of the tree, abstracting a sub-term as explained at the end of Section 6. In this view, trees in the dynamic unit and graphs in the static one are then active entities evolving autonomously and not standard data structures.

Each operator defined in this family manipulates the structure of actual term, modifying (creating or destroying) affinities among automata.

8.3. Static Family

As in dynamic unit, graph representation is realised by affinities among automata established by state transitions. Operators defined in the static unit preserve graphs structure, possibly adding new graphs into the unit. Adding new structures (i.e. creating terms) establishes new affinities, preserving the previous ones.

8.4. Some Remarks

Machine functions are expressed by means of transition rules and synchronised communication of automata families.

Homogeneous automata have the same state transition rules. Complex algorithms are therefore distributed onto single automata transitions (i.e. split in area and time). This supports a concurrent design style and allows the use of a single falsification strategy for all automata in a family.

We define automata behaviour for all possible input patterns. The problem of achieving well defined algorithms (i.e. whose behaviour is defined on the entire input domain) turns into the problem of achieving well defined transition tables. This kind of context dependent calculus, managing local information with possibly 'global' scope, makes qualitatively meaningless to realise communications inside a family by means of network topologies different the complete one. Complete topology disregards realistic requirements, being realistically non-scalable. On the other hand, every incomplete connection topology (i.e. with a number of arcs per vertex less than the total number of vertexes -1) is not suited to one-step evaluation of operators with context dependent scope. From a qualitative point of view, incomplete topologies are equivalent with respect to this calculus.

The complexity of context dependent computations may require a fine characterisation of automaton state, thus exploding transition table dimension. It is often possible, however, to factorize the transition table (making

its dimension independent from family population), splitting a single step transition into multiple dummy transitions.

9. Conclusions

This work is a first attempt towards the solution to the problem of system testability. To this aim we faced with the problem, in its maybe most difficult formulation, of the testability of a machine implementing an abstract theory.

We pointed out that a theory cannot be completely implemented by real machines because of their intrinsic finiteness. Thus the realistic assumption is made: the machine model is at most a finite restriction of the theory, which however contains the essential and specific features of the theory.

We defined the necessary features to gain machine testability, assuming their validity can be extended to machines implementing less formal specifications than a theory usually requires.

We focused on lambda-calculus to give an example of the way a theory is analysed in order to identify one of its subsets. Though finite, this has to be rich enough to preserve the essential features of calculus, such as context dependency, constructive definition of terms, etc.. This kind of analysis naturally leads to the definition of a high level architecture. We sketched a realisation by means of a distributed automata architecture, showing how it matches the specifications coming from realistic model of lambda-calculus (abstraction on terms, locality, etc.).

The assumption of the realistic model, abstracting on technologies or particular design choices, forces to contemplate requirements of testable machines from the very first design stages. This may lead to re-define specifications themselves. Moreover, this design process supports machine architectures reflecting calculus semantics. It is thus possible to build machines, whose internal evolution is directly interpretable and foreseeable by the realistic calculus model.

By abstracting as much as possible on low level architectural choices, the work has shown the implementation limits of lambda-calculus and suggested the possibility to apply the obtained results to other expression theories.

The choice of broadcast automata as low level architecture allowed us to investigate their expressiveness. This analysis resulted in the definition of a BA-family architecture as an extension (BORDONI et al., 1997).

A simulator of the lambda-machine has been developed in C language and a first prototype of the system has been implemented by using FPGA technology.

References

[1] ALDERIGHI, M. – MAZZEI, R. P.G. – SECHI, G. R. – TISATO, F. (1997a): Broadcast Automata: a Parallel Scalable Architecture for Prototypal Embedded Processors for Space Applications, *Proc. of the Thirtieth Annual Hawai'i International Conference on System Sciences*, (Maui, Hawaii, January 7-10, 1997), Vol. V, IEEE Press, pp. 208–217.

[2] ALDERIGHI, M. – CASINI, F. – MAZZEI, R. P. G. – SECHI, G. (1997b): Broadcast Automata: a Computational Model for Massively Parallel Symbolic Processing, in this volume, 1997.

[3] ASPERTI, A. – LONGO, G. (1991): Categories, Types, and Structures, The MIT Press, Cambridge, MA, 1001.

[4] BACKUS, J. (1978): Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs, *Communication of the ACM*, Vol. 21, N. 8, 1978, pp. 613–641.

[5] BARENDREGT, H. P. (1984): The Lambda Calculus, its Syntax and Semantics, North Holland, revised edition 1984.

[6] BORDONI, A. – MOJOLI, G. A. (1997): Macchine ad automi cellulari plurifamiglia su topologie generiche: una estensione del modello broadcast automata, IFCTR Internal Report, June 1997.

[7] COUSINEAU, G. – COURIEN, P. L. – MAUNY, M. (1987): The Categorical Abstract Machine, *Science of Computer Programming*, Vol. 8, 1987, pp. 173–202.

[8] FAIRBAIRN, J. – WRAY, S. (1987): Tim: a Simple Lazy Abstract Machine to Execute Supercombinators, *Proc. of the International Conference on Functional Programming Languages and Computer Architecture, (FPLCA'87)*, LNCS N. 274, (G. Kan Ed.), (Portland, Oregon, USA, September 1987), Springer-Verlag, pp. 34–45.

[9] GENTZEN, G. (1969): Investigation into Logical Deduction, The Collected Papers of Gerhard Gentzen, M. E. Szabo, 1969.

[10] GIRARD, J. Y. – LAFONT, Y. – TAYLOR, P. (1989): Proofs and Types, *Cambridge Tracts in Theoretical Computer Science*, Vol. 7, Cambridge University Press, 1989.

[11] HANNAN, J. – MILLER, D. (1990): From Operational Semantics to Abstract Machines, *Math. Struct. in Comp. Science*, Vol. 2, 1992, Cambridge University Press, pp. 415–459.

[12] HANNAN, J. (1991): Making Abstract Machines Less Abstract, *Proc. of the Fifth ACM Conference on Functional Programming Langugaes and Computer Architectures*, 1991, pp. 618–635.

[13] HINDLEY, J. R. (1986): Introduction to Combinators and Lambda-Calculus, Cambridge University Press, 1986.

[14] The ML Handbook, Inria Technical Report, 1984.

[15] LAFONT, Y. (1988): The Linear Abstract Machine, *Theoretical Computer Science*, Vol. 59, 1988, pp. 157–180.

[16] LANDIN, P. J. (1964): The Mechanical Evaluation of Expressions, *Computer Journal*, Vol. 6, 1964, pp. 308–320.

[17] MAGNUSSON, L. – NORDSTRÖM, B. (1994): The ALF Proof Editor and its Proof Engine, Report of the University of Göteborg/Chalmers, Sweden, 1994.

[18] NORDSTRÖM, B. – PETERSSON, K. – SMITH, J. M. (1990): Programming in Martin-Löf's Type Theory, An Introduction, Oxford Science Publications, 1990.

[19] PLOTKIN, G. (1975): Call-by-Name, Call-by-Value and the Lambda-Calculus, *Theoretical Computer Science*, Vol. 1, N. 2, 1975, pp. 125–159.

[20] POPPER, P. R. (1970): Logica della scoperta scientifica, Einaudi, 1970.

[21] REVESZ, G. E. (1988): Lambda-Calculus, Combinators, and Functional Programming, Cambridge University Press, 1988.

[22] TURNER, D. A. (1979): A New Implementation Technique for Applicative Languages, *Software Practice and Experience*, Vol. 9, 1979, pp. 31–49.