An Abstract Interpretation Framework for (almost) Full Prolog*

Baudouin Le Charlier† Sabina Rossi‡ Pascal Van Hentenryck§

†University of Namur, 21 rue Grandgagnage, B-5000 Namur (Belgium)
‡University of Padova, via Belzoni 7, I-35131 Padova (Italy)
§Brown University, Box 1910, Providence, RI 02912 (USA)

Abstract

A novel abstract interpretation framework is introduced, which captures the Prolog depth-first strategy and the cut operation. The framework is based on a new conceptual idea, the notion of substitution sequences, and the traditional fixpoint approach to abstract interpretation. It broadens the class of analyses that are amenable in practice to abstract interpretation and refines the precision of existing analyses[1].

1 Introduction

Abstract interpretation has been shown to be a valuable tool to realize high-performance implementations of Prolog [18, 19]. Yet, traditional abstract interpretation frameworks (e.g. [3, 13, 14]) usually ignore many features of Prolog, such as the depth-first search strategy and the cut operation. Also, although these frameworks are very valuable - since they allow to perform accurately many useful practical analyses such as types, modes, and sharing - their limitations become more evident, as the technology matures. In particular, they lead to the following two inconvenients.

1. The precision of the analysis is inherently limited for some classes of programs. A typical example is the definition of multi-directional procedures, using cuts and meta-predicates to select among several versions. Ignoring the depth-first search strategy and the cut prevents the compiler from performing various important compiler optimizations such as dead-code elimination [2].

*Partly supported by the Office of Naval Research under grant N00014-91-J-4052 ARPA order 8225 and the National Science Foundation under grant numbers CCR-9357704 and a NSF National Young Investigator Award.

[1] Only parts of the theoretical results are presented here. The reader is referred to the technical report version [11] for a comprehensive coverage and an accurate discussion on related works.

2. The existing frameworks are not expressive enough to capture certain analyses in their entirety. A typical example is determinacy analysis, where existing approaches either resort to special purpose proofs (e.g. [16]) or their frameworks ignore certain aspects of the analysis, e.g. the cut and/or how to obtain the determinacy information from input/output patterns (e.g. [7, 6]).

This paper proposes a step in overcoming these limitations. A novel abstract interpretation framework is introduced, which captures the depth-first seach strategy and the cut operation (only dynamic predicates such as `assert/retract` are ignored). The key conceptual idea underlying the framework is the notion of substitution sequences which models the successive answer substitutions of a Prolog goal. This notion enables the framework to deduce and reason about information not available in most frameworks, such as sure success and failure, the number of solutions, and/or termination, broadening the class of applications amenable to abstract interpretation and improving the accuracy of existing analyses.

The main technical contribution of this paper is to show how to apply the traditional fixpoint approach [5] to the conceptual idea. A main difficulty lies in the fact that the abstract semantics cannot simply be defined as the least fixpoint of the abstract transformation obtained from the collecting semantics, since the least fixpoint of the transformation obtained by "lifting" the concrete semantics to sets of substitution sequences is not a consistent approximation of the concrete semantics. The notion of pre-consistent postfixpoint is introduced to remedy this problem. The practical consequences of this formalization are discussed and include the need for so-called upper-closed abstract domains and a special form of widening in the abstract interpretation algorithm.

The framework is motivated by computational considerations and its practicability and simplicity have been demonstatred on a cardinality analysis described in [2]. The rest of the paper is organized as follows. Section 2 motivates the paper through extremely simple examples and gives an overview of the framework. Section 3 is an informal presentation of the main technical difficulties on a single example as well as the adopted solutions. Section 4 sketches the concrete semantics. Section 5 and 6 contain respectively the specification of the abstract operations and the abstract semantics. Section 7 discusses the abstract interpretation algorithm.

2 Overview of the Framework

Let us first illustrate two simple examples which are not handled well by existing abstract interpretation frameworks. Consider the program:

$$p(a). \quad p(x) :- q(x). \quad q(b). \quad q(c).$$

Suppose that we are interested in determinacy analysis of p/1 called with a ground argument. Examinations of the clauses in isolation will not determine the determinacy of the goal. This was recognized in several places (e.g. [7, 6]) which proposes to use input/output patterns to remedy the problem. However, these works focus on determining the patterns and cannot integrate all aspects of the analysis in a single abstract interpretation framework. As a consequence, they need special-purpose

proofs for the final part of the analysis losing the simplicity of the abstract interpretation framework. Our framework handles all aspects of the analysis in a single framework. Moreover these works do not take control (depth-first search and the cut) into account reducing the precision of the analysis. In fact, consider the extension of the previous program:

$$\mathtt{r}(x) : -\mathtt{p}(x), !. \qquad \mathtt{r}(d).$$

and assume that r/1 is called with a variable. Abstract interpretation frameworks ignoring the search rule and the cut operation, when instantiated with a type domain (e.g. [8, 4]), would conclude that the goal has an answer from $\{a, b, c, d\}$. Using our framework, it is possible to design an analysis concluding that r/1 only produces the element a. We now describe informally the basic ideas on how to obtain such an analysis.

Concrete Semantics The starting point of our approach is a concrete semantics which associates with a program P a total function from the set of pairs $\langle \theta, p \rangle$, where θ is a substitution and p is a n-ary predicate symbol, to the set of substitution sequences S, as follows. The sequence S corresponding to a pair $\langle \theta, p \rangle$, noted $\langle \theta, p \rangle \longmapsto S$, models the sequence of computed answer substitutions (e.g. [12]) produced by the execution of $p(x_1, \ldots, x_n)\theta$. The sequence S can have different shapes. If the execution terminates (producing m computed answer substitutions), S is a *finite* sequence $< \theta_1, \ldots, \theta_m >$. If the execution produces m computed answer substitutions and then enters into an infinite loop, then S is an *incomplete* sequence $< \theta_1, \ldots, \theta_m, \bot >$, where $\bot$ models non termination [1]. Finally, if the execution produces an infinite number of computed answer substitutions, then S is an *infinite* sequence $< \theta_1, \ldots, \theta_i, \ldots >$ $(i \in \mathbf{N})$. We note $SUBST(S)$ the set of substitutions in S.

Abstract Semantics The abstract semantics works with description of sequences called abstract sequences. It associates with a program a total function which, given a pair $\langle \beta, p \rangle$ (where β is an abstract substitution), returns an abstract sequence B, whose informal semantics can be described as follows: "The execution of $p(x_1, \ldots, x_n)\theta$ with θ satisfying the property β produces a substitution sequence S satisfying the property described by B."
It is important to realize that abstract domains for sequences need not be much more complicated than traditional abstract domains. We illustrate this with two examples.

Abstract Domain 1: An abstract sequence B is of the form $\langle \beta, m, M, t \rangle$ where β is an abstract substitution, $m \in \mathbf{N}$, $M \in \mathbf{N} \cup \{\infty\}$, $t \in \{snt, st, pt\}$. B describes the set of substitution sequences S such that any substitution θ in S is described by β and the number of elements of S, excluding $\bot$, is not smaller than m and not greater than M. Additionally, the sequences S are all finite if $t = st$, they are all incomplete or infinite if $t = snt$, they can have any form if $t = pt$ (snt means "sure non termination", st means "sure termination" and pt stands for "possible termination"). Using this abstract domain on our first program, the abstract semantics

defines $\langle \mathtt{p}(\{a, b, c\}), 0, 1, st) \rangle$ as the result of a query p*(ground)*. Note that the domain is not too complex computationally.

Abstract Domain 2: The first abstract domain does not achieve maximal precision on the second program when considering the query r*(variable)*. It produces the result $\langle \mathtt{r}(\{a, b, c\}), 0, 1, st) \rangle$. A more precise domain consists of abstract sequences B of the form $\langle < \beta_1, \ldots, \beta_m >, \beta, m, M, t \rangle$, where m, M and t are given the same meaning as above. Each S described by B must be of the form $< \theta_1, \ldots, \theta_m >:: S'$ where for all $i \in \{1, \ldots, m\}$, θ_i is described by β_i and each substitution in S' is described by β (:: denotes the usual concatenation operation on sequences). Using this domain, the abstract semantics defines $\langle < \mathtt{p}(\{a\}), \mathtt{p}(\{b\}), \mathtt{p}(\{c\}) >, \mathtt{p}(\{\}), 3, 3, st) \rangle$ as the result of p*(variable)* and $\langle < \mathtt{r}(\{a\}) >, \mathtt{r}(\{\}), 1, 1, st) \rangle$ as the result of r*(variable)*. The new domain is likely to be computationally reasonable, since there are few situations where a large number of abstract substitutions will be maintained.

Abstract Interpretation Algorithm The last step of the analysis is the computation of the abstract semantics with extensions of existing algorithms such as *GAIA* [10] and *PLAI* [15].

3 Technical Difficulties and Adopted Solutions

The foundation of this work is the fixpoint approach to abstract interpretation [5]. Starting from a concrete semantics, we try to define a collecting semantics, an abstract semantics approximating the collecting semantics, and an algorithm to compute part of the abstract semantics. Applying this approach to the above informal ideas leads to some novel theoretical and practical problems. The main problem is that the abstract semantics can no longer be defined as the least fixpoint of the basic transformation obtained by "lifting" the concrete semantics to sets of substitution sequences. In this section, we illustrate these problems and their proposed solutions.

Concrete Semantics Consider the following program

```
repeat.    repeat : -repeat.
```

The concrete semantics of this program maps the input $< \varepsilon, \mathtt{repeat} >$, where ε is the empty substitution, to the infinite sequence $S =< \varepsilon, \ldots, \varepsilon, \ldots >$. This comes from the fact that the result S is described as the least fixpoint of a transformation $\tau_1 : PSS \rightarrow PSS$ (where PSS denotes the set of substitution sequences):

$$\tau_1 S =< \varepsilon >:: S.$$

Operationally, this expresses the fact that the first clause succeeds producing the result ε; whereas the second clause succeeds exactly as many times as the recursive call, producing the same sequence of results. PSS can be endowed with the following ordering: $S_1 \sqsubseteq S_2$ iff either $S_1 = S_2$ or there exist $S, S' \in PSS$ such that $S_1 = S ::< \bot >$ and $S_2 = S :: S'$. PSS is then a pointed *cpo* with minimal element $< \bot >$.

τ_1 is continuous and has a least fixpoint which is computed as follows: $S_0 =< \bot >$, $S_{i+1} =< \varepsilon >:: S_i =< \varepsilon, \ldots, \varepsilon, \bot >$ (with i occurrences of ε), and $lfp(\tau_1) = \sqcup_{i=0}^{\infty} S_i$ $=< \varepsilon, \ldots, \varepsilon, \ldots >$ as expected.

Collecting Semantics The technical problems arise when we "lift" the semantics to sets of substitution sequences. The "collecting" semantics associates with the program the transformation $\tau_2 : \wp(PSS) \rightarrow \wp(PSS)$ defined by

$$\tau_2\Sigma = \{< \varepsilon >:: S \ : \ S \in \Sigma\}.$$

$\wp(PSS)$ is a complete lattice for set inclusion and τ_2 is monotonic. However, $lfp(\tau_2)$ is not a consistent approximation of $lfp(\tau_1)$ (i.e. $lfp(\tau_1) \notin lfp(\tau_2)$), since $lfp(\tau_2)$ is the empty set. Note however that τ_2 is consistent with respect to τ_1 in the following sense: for all $S \in PSS$ and for all $\Sigma \in \wp(PSS)$, $S \in \Sigma$ implies $\tau_1(S) \in \tau_2(\Sigma)$.

The first cause of inconsistency of $lfp(\tau_2)$ is that S_0, the first iterate in the Kleene sequence for $lfp(\tau_1)$, obviously does not belong to the first iterate of the Kleene sequence for $lfp(\tau_2)$ (which is empty). In order to get a consistent approximation of $lfp(\tau_1)$, we may attempt to build another sequence of sets of substitution sequences as follows:

$$\Sigma_0 = \{< \bot >\}, \ \Sigma_{i+1} = \tau_2\Sigma_i = \{< \varepsilon, \ldots, \varepsilon, \bot >\} \ (i \geq 0).$$

The problem is that this sequence is not increasing with respect to inclusion.

This new problem could possibly be solved by using another ordering on (some subset of) $\wp(PSS)$. This ordering should in a way combine the ordering on PSS and inclusion in $\wp(PSS)$. The traditional solution to this problem in denotational semantics consists in using a power domain construction (e.g. [17]). Although this solution is elegant theoretically, it is somewhat heavy for an abstract interpretation framework which should lead to efficient implementations. We adopted a solution which is less natural from a denotational standpoint but leads to effective analyses as demonstrated in the work [2]. The solution is best presented in three steps.

First, τ_2 is replaced by a transformation τ_3:

$$\tau_3\Sigma = \Sigma \cup \tau_2\Sigma.$$

τ_3 is *extensive* (i.e. $\Sigma \subseteq \tau_3\Sigma$ for all Σ). In addition, the sequence defined by $\Sigma_0 = \{< \bot >\}$ and $\Sigma_{i+1} = \tau_3\Sigma_i$ is increasing and its limit is the set:

$$\Sigma_\infty = \bigcup_{i=0}^{\infty} \Sigma_i = \{< \bot >, < \varepsilon, \bot >, \ldots, < \varepsilon, \ldots, \varepsilon, \bot >, \ldots\}.$$

Σ_∞ contains the entire Kleene sequence for $lfp(\tau_1)$ but still not $lfp(\tau_1)$ itself.

The second step is thus to complete the sets of substitution sequences containing increasing chains of sequences (with respect to $\sqsubseteq$) with their limits. Sets of substitution sequences so completed are called *upper-closed* and we denote by CSS the set of such upper-closed sets. τ_2 and τ_3 can be redefined over CSS. The upper bound operation $\bigsqcup$ in CSS is no longer $\bigcup$: it adds to the union the limit of every chain in the union. Applying the new construction to τ_3 leads to the result $\Sigma_\infty = \bigsqcup_{i=0}^{\infty} \Sigma_i$ which contains all non finite sequences of empty substitutions.

The last step of our construction consists in refining this correct but imprecise result. Instead of starting the iteration with τ_3, τ_2 is used during an arbitrary number of steps before switching to τ_3. Since each iterate for τ_2 contains the corresponding iterate for τ_1, switching to τ_3 after i steps guarantees that the set Σ_∞ contains all iterates from the i-th and also the limit, since sets are upper-closed. In the above example, we deduce that `repeat` produces at least i results.

Abstract Computation The construction can be adapted to the abstract semantics by using consistent abstractions of τ_2 and τ_3. However, if the abstract domain is not noetherian, a widening operation must be used instead of the upper bound operation to ensure the finiteness of the analysis. Let us consider this last case. Consider an abstract domain of abstract substitution sequences, ASS, with a concretization function $Cc : ASS \rightarrow CSS$ and with an element B_0 such that $< \bot > \in Cc(B_0)$. Consider also an abstract version τ_4 of τ_2, i.e.

$$\forall S \in PSS \ \ \forall B \in ASS : S \in Cc(B) \Rightarrow \tau_1 S \in Cc(\tau_4 B).$$

The computation in the abstract domain iterates τ_4 for j steps:

$$B_{i+1} = \tau_4 B_i \ (0 \leq i \leq j).$$

Then, *unless a fixpoint has already been reached*, the computations "jumps" to a value B_w such that $B_j \leq B_w$ and $\tau_4 B_w \leq B_w$ (i.e., B_w is a postfixpoint of τ_4).

The process is sound for the following reason. Let S_j be the iterates to $lfp(\tau_1)$. Since $S_0 = \langle\bot\rangle \in Cc(B_0)$ and τ_4 is consistent, $S_j \in Cc(B_j)$ by induction. Since $B_j \leq B_w$ and B_w is a postfixpoint, $S_k \in Cc(B_w)$ for all $k \geq j$. In fact if for some k, $S_k \in Cc(B_w)$ then $S_{k+1} \in Cc(\tau_4 B_w)$, by consistency of τ_4, and hence, $S_{k+1} \in Cc(B_w)$ by $\tau_4 B_w \leq B_w$ and monotonicity of Cc. Finally, since $Cc(B_w)$ is upper-closed, $lfp(\tau_1) \in Cc(B_w)$.

To illustrate the process on a concrete example, consider the first abstract domain, dropping the abstract substitution part since it is useless. τ_4 is defined by $\tau_4\langle m, M, t\rangle = \langle m+1, M+1, t\rangle$ and $B_0 = \langle 0, 0, snt\rangle$. The first iterations give $B_j = \langle j, j, snt\rangle$. To get a postfixpoint, the second j is replaced by ∞ to obtain $B_w = \langle j, \infty, snt\rangle$, since $\tau_4 B_w = \langle j+1, \infty, snt\rangle \leq B_w$. B_w is a consistent approximation of $lfp(\tau_1)$ and expresses that at least j substitutions are generated and that the procedure surely loops. We do not know however if it loops after giving a finite number of substitutions or if it produces an infinite number of substitutions.

Theoretical and Practical Implications The above construct implies that the abstract semantics can no longer be defined as the least fixpoint of the abstract transformation obtained by abstracting the collecting semantics. The abstract semantics is defined as certain postfixpoints of the abstract transformation (see the definition of pre-consistent set of abstract tuples later on).

In practice, the construct imposes two requirements on the abstract domain. First, it is necessary to make sure that the concretization function only returns upper-closed

sets. This requirement, which is satisfied by our two abstract domains, does not seem to be too restrictive in practice. Second, the designer needs to decide when to apply the widening operation. This is of course domain-dependent.

4 Concrete Semantics

Space restrictions forbid us to include the concrete semantics in the paper. Since it is not essential for the comprehension of the abstract semantics, in this section we simply sketch it. The concrete semantics is a fixpoint semantics defined on normalized programs [3] such that clause heads are of the form $p(x_{i_1}, \ldots, x_{i_n})$ and bodies contain atoms of the form $p(x_{i_1}, \ldots, x_{i_n})$, $x_i = x_j$, $x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$, and !. To simplify the traditional problems with renaming, we use two sets of variables and substitutions [10]. (Program) substitutions (denoted by θ) are of the form $\{x_1/t_1, \ldots, x_n/t_n\}$, where the t_i are terms and the x_i are (so-called) *program* variables (or parameters). We assume another infinite (disjoint) set of (so-called) *standard* variables. The t_i's may only contain standard variables. By definition, $dom(\theta) = \{x_1, \ldots, x_n\}$ and $codom(\theta)$ is the set of variables in the t_i's. We also use *standard* substitutions which are substitutions in the usual sense containing only standard variables. They are denoted by σ possibly subscripted. *mgu*'s are standard substitutions. The composition $\theta\sigma$ of a program substitution with a standard substitution is defined in a non standard way by $\theta\sigma = \{x_1/t_1\sigma, \ldots, x_n/t_n\sigma\}$. We note PS the set of program substitutions.

The concrete semantics uses objects of the form $\langle\theta, p\rangle$, $\langle\theta, pr\rangle$, $\langle\theta, c\rangle$, and $\langle\theta, g\rangle$, where p, pr, c and g are respectively a predicate name, a procedure, a clause, and the body or a prefix of the body of a clause. It also uses substitution sequences and objects of the form $\langle S, cf\rangle$, where S is a substitution sequence and $cf \in \{cut, nocut\}$ (we denote $CSSC$ the set of such elements). Objects of the form $\langle\theta, p\rangle$ and $\langle\theta, pr\rangle$ (where pr is the procedure defining p) are mapped to the substitution sequence which models the sequence of answer substitutions for $p(x_1, \ldots, x_n)\theta$. Objects of the form $\langle\theta, c\rangle$ and $\langle\theta, g\rangle$ are mapped to objects of the form $\langle S, cf\rangle$, where cf indicates whether the execution of the clause c or of the goal g has been cut. Assuming an underlying program P, we note by $\langle\theta, p\rangle \longmapsto S$ the fact that the concrete semantics of P maps $\langle\theta, p\rangle$ to S.

5 Abstract Operations

We assume the existence of three *cpos*: AS, ASS and $ASSC$. Elements of AS are called abstract substitutions and denoted by β. Elements of ASS are called abstract sequences and denoted by B. Elements of $ASSC$ are called abstract sequences with cut information and denoted by C. The meaning of these abstract objects is given through monotonic concretization functions: $Cc : AS \to CS$, $Cc : ASS \to CSS$ and $Cc : ASSC \to CSSC$. $CS = \wp(PS)$, CSS is the set of sets of substitution sequences which are upper-closed. $CSSC$ is similarly defined but increasing chains only contain substitution sequences with identical cut information. CS, CSS and $CSSC$ are ordered by inclusion. Each object o in AS, ASS and $ASSC$ has a domain, $dom(o)$, which is the common domain of all program substitutions in its concretization.

In this section we motivate and specify by a consistency condition each abstract operation. Many of these operations are identical or simple generalizations of operations described in [9, 10], which were themselves inspired by [3]. Other are simple "conversion" operations between the three different domains described above. The newer operations are *CONC*, *AI-CUT*, *EXTGS* and they are explained in detail since they contain the main originality of our framework. Reference [2] proposes an implementation of all operations on a particular abstract domain. The abstract operations are the following.

Concatenation of Abstract Sequences: $CONC(\beta,C,B)=B'$. Let pr be a procedure of the form $c_1, \ldots, c_n$ $(n \geq 1)$. A *suffix* of pr is any sequence of clauses $c_i, \ldots, c_n$ $(1 \leq i \leq n)$. Operation *CONC* is used to "concatenate" (at the abstract level) the result C of a clause c_i with an abstract sequence B resulting from "concatenating" the results of $c_{i+1}, \ldots, c_n$ $(1 \leq i < n)$. It is assumed that all results are produced for the same abstract input substitution β. β is added as an extra parameter in order to improve the accuracy of the operation.

In order to express the consistency conditions for the operation *CONC*, "concatenation" of concrete sequences needs to be defined first. Consider two sequences S_1 and S_2 without cut information. S_1 stands for the result of c_i and S_2 stands for the (combined) result of $c_{i+1}, \ldots, c_n$. If execution of c_i terminates, then suffix $c_{i+1}, \ldots, c_n$ is executed. Otherwise $c_{i+1}, \ldots, c_n$ is not executed. Therefore, the combined result, $S_1 \Box S_2$, of $c_i, \ldots, c_n$ is defined by

$$S_1 \Box S_2 \quad = \quad \begin{array}{ll} S_1 :: S_2 & \text{if } S_1 \text{ is finite (i.e. neither incomplete nor infinite),} \\ S_1 & \text{otherwise.} \end{array}$$

The definition can be extended to sequences with cut information. If no cut is executed by computing c_i (because there are no cuts or the execution of c_i fails or loops before reaching a cut), then the previous reasoning applies. Otherwise, suffix $c_{i+1}, \ldots, c_n$ is not executed. In the first case, the result of c_i is $\langle S_1, nocut\rangle$, while, in the second case, the result is $\langle S_1, cut\rangle$. So, the combined result $\langle S_1, cf\rangle \Box S_2$ of $c_i, \ldots, c_n$ is defined by

$$\langle S_1, cf\rangle \Box S_2 \quad = \quad \begin{array}{ll} S_1 \Box S_2 & \text{if } cf = nocut, \\ S_1 & \text{if } cf = cut. \end{array}$$

Operation *CONC* performs the concatenation of abstract sequences, i.e. of descriptions of sets of sequences, and is defined as follows (recall that $CONC(\beta,C,B)=B'$)[2]:

$$\left.\begin{array}{l} \theta \in Cc(\beta), \\ \langle S_1, cf\rangle \in Cc(C), \\ S_2 \in Cc(B), \\ \forall \theta' \in SUBST(S_1) \cup SUBST(S_2) : \theta' \leq \theta \end{array}\right\} \Rightarrow \langle S_1, cf\rangle \Box S_2 \in Cc(B').$$

Since β represents many different input substitutions, C and B may contain *incompatible* substitution sequences, i.e. sequences containing substitutions which are

[2] In order to enhance readability of the specifications, it is assumed that all free symbols are implicitely universally quantified and range over a domain which is "obvious" from the context.

not all instances of the same input substitution. Concatenations of incompatible substitution sequences are removed by the last condition, since they do not correspond to any actual execution. ($\theta' \leq \theta$ means that θ' is more instantiated than θ.)

Abstract Unification of two program variables: $AI\text{-}VARS(\beta) = B'$. This operation unifies variables x_1 and x_2 called with input abstract substitution β. It is similar to operation *AI-VAR* of [9, 10] but returns an abstract sequence instead of an abstract substitution. Clearly, $Cc(B')$ should only contain *finite* sequences of length 0 or 1. Let $Smgu = mgu(x_1\theta, x_2\theta)$.

$$\left.\begin{array}{l}\theta \in Cc(\beta),\\ \sigma \in Smgu\end{array}\right\} \Rightarrow <\theta\sigma> \in Cc(B'); \qquad \left.\begin{array}{l}\theta \in Cc(\beta),\\ \emptyset = Smgu\end{array}\right\} \Rightarrow <> \in Cc(B').$$

Abstract Unification of a variable and a functor: $AI\text{-}FUNCS(\beta, f) = B'$. This operation is similar to the previous one: it unifies the variable x_1 with $f(x_2, \ldots, x_n)$ called with input β. Let $Smgu = mgu(x_1\theta, f(x_2, \ldots, x_n)\theta)$.

$$\left.\begin{array}{l}\theta \in Cc(\beta),\\ \sigma \in Smgu\end{array}\right\} \Rightarrow <\theta\sigma> \in Cc(B'); \qquad \left.\begin{array}{l}\theta \in Cc(\beta),\\ \emptyset = Smgu\end{array}\right\} \Rightarrow <> \in Cc(B').$$

Abstract Treatment of the Cut: $AI\text{-}CUT(C) = C'$. Let g be the sequence of literals before a cut (!) in a goal. Execution of g for a given input substitution θ either fails or loops without producing any result, or produces one or more results before terminating, looping or producing results for ever. Execution of the goal $g, !$ also fails or loops without producing results in the first case but, in the second case, it produces exactly one result (the first result of g) and then stops. At the abstract level, C represents a set of substitution sequences produced by g, while C' represents the corresponding set of substitution sequences produced by $g, !$. Clearly the sequences in $Cc(C')$ should be obtained by "cutting" the sequences in $Cc(C)$ after their first element if they have one. Hence, the following specification:

$$\begin{array}{l}\langle <>, cf\rangle \in Cc(C) \Rightarrow \langle <>, cf\rangle \in Cc(C');\\ \langle <\bot>, cf\rangle \in Cc(C) \Rightarrow \langle <\bot>, cf\rangle \in Cc(C');\\ \langle <\theta> :: S, cf\rangle \in Cc(C) \Rightarrow \langle <\theta>, cut\rangle \in Cc(C').\end{array}$$

We now turn to the projection and extension operations. The first and the third are the same as in our previous papers [9, 10]. The second one is a simple generalization of an existing one to sequences. The fourth one is a more complex generalization and we explain it in detail.

Extension at Clause entry: $EXTC(c, \beta) = \beta'$. This operation extends the input β on variables $\{x_1, \ldots, x_n\}$ of the head of the clause c to all variables $\{x_1, \ldots, x_m\}$ $(m \geq n)$ in c.

$$\left.\begin{array}{l}\theta \in Cc(\beta),\\ y_1, \ldots, y_{m-n} \text{ are distinct standard variables,}\\ y_1, \ldots, y_{m-n} \notin codom(\theta)\end{array}\right\} \Rightarrow \{x_1/x_1\theta, \ldots, x_n/x_n\theta, x_{n+1}/y_1, \ldots, x_m/y_{m-n}\} \in Cc(\beta').$$

Restriction at Clause Exit: $RESTRC(c, C) = C'$. Consider the same notations as above and assume that the execution of the body of c, for the input β', produces the abstract sequence with cut information C. Operation *RESTRC* simply restricts C on all variables in c to the variables $\{x_1, \ldots, x_n\}$ in the head.

$$<\theta_1, \ldots, \theta_i, \ldots>, cf) \in Cc(C) \Rightarrow (<\theta_{1|D}, \ldots, \theta_{i|D}, \ldots>, cf) \in Cc(C').$$

Restriction before a Call: $RESTRG(l, \beta) = \beta'$. Let l be a literal $p(x_{i_1}, \ldots, x_{i_n})$ (or $x_{i_1} = x_{i_2}$ $(n = 2)$ or $x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$) (or any other built-in using variables $x_{i_1}, \ldots, x_{i_n}$). This operation express the input β for l in terms of its formal parameters $\{x_1, \ldots, x_n\}$.

$$\theta \in Cc(\beta) \Rightarrow \{x_1/x_{i_1}\theta, \ldots, x_n/x_{i_n}\theta\} \in Cc(\beta').$$

Extension of the Result of a Call: $EXTGS(l, C, B) = C'$ This operation is rather complex and we first motivate it through the correspondence between the concrete and abstract executions. We assume the same notations as for *RESTRG*, that l occurs in the body of a clause c and that g is the sequence of literals before l in the body.

In the concrete semantics, execution of g for an input substitution θ produces a sequence (with cut information) $\langle S, cf\rangle$. Then l is executed for each substitution θ_i of S, producing a new sequence S_i for each θ_i. The "result" of g, l is the sequence $S_1 \Box \ldots \Box S_i \ldots$.

At the abstract level, C stands for a set of possible $\langle S, cf\rangle$'s while B stands for (a superset of) all corresponding S_i's. Because of the abstraction, the mapping between each S and its corresponding S_i's is lost as well as the ordering of the S_i's. Operation *EXTGS* has to reestablish this mapping as best as possible by a kind of backward unification.

Note that, in the above (concrete) concatenation, there can be infinitely many S_i's and the definition of $\Box$ must be extended as follows: $\Box_{k=1}^{0} S_k = <>$; $\Box_{k=1}^{i+1} S_k = (\Box_{k=1}^{i} S_k)\Box S_{i+1}$ $(i \geq 0)$; $\Box_{k=1}^{\infty} S_k = \sqcup_{i=0}^{\infty}((\Box_{k=1}^{i} S_k)\Box < \bot >)$.
It can be verified that this definition fits the intuition in all cases. For instance, if one of the S_i is incomplete or infinite, subsequent sequences are ignored. $\Box_{k=1}^{\infty} <>=< \bot >$ which expresses that the computation of an infinite number of sequences (albeit all empty) never terminates.

More technically, the result B of the execution of l called with C is obtained by 1) extracting the substitution part β of C (the sequence structure is forgotten), 2) applying *RESTRG* to β obtaining β', 3) executing l with input β'. Therefore, B is an abstract sequence on $\{x_1, \ldots, x_n\}$ and we have to reexpress it on $\{x_{i_1}, \ldots, x_{i_n}\}$ while combining it with C. The precise specification is as follows. $NELEM(S)$ stands for the number of elements in S. $NELEM(S) = NSUBST(S) + 1$ if S is incomplete; otherwise, $NELEM(S) = NSUBST(S)$.

$$\left.\begin{array}{l}\langle S, cf\rangle \in Cc(C),\\ (\forall k : 1 \le k \le NSUBST(S):\\ \theta_k \text{ is the } k\text{-th substitution of } S,\\ \theta'_k = \{x_1/x_{i_1}\theta_k, \ldots, x_n/x_{i_n}\theta_k\},\\ S'_k \in Cc(B),\\ S'_k =< \theta'_k\sigma_{k,1}, \ldots, \theta'_k\sigma_{k,j}, \ldots >,\\ S_k =< \theta_k\sigma_{k,1}, \ldots, \theta_k\sigma_{k,j}, \ldots >)\end{array}\right\} \Rightarrow \langle \Box_{k=1}^{NELEM(S)} S_k, cf\rangle \in Cc(C').$$

In order to prevent introduction of undesired variable sharing in the result, we can also specify that no substitution $\sigma_{k,j}$ introduces "new" variables already in $codom(\theta_k)$ but not in $codom(\theta'_k)$. Formally: $dom(\sigma_{k,j}) \subseteq codom(\theta'_k)$ and $(codom(\theta_k) \backslash codom(\theta'_k)) \cap codom(\sigma_{k,j}) = \emptyset\ \forall k, j$.

Finally, we need three less important conversion operations.

$SEQ(C) = B'$. This operation forgets the cut information in C. It is applied to the result of the last clause of a procedure before combining this with the result of the other clauses.

$$\langle S, cf\rangle \in Cc(C) \Rightarrow S \in Cc(B').$$

$SUBST(C) = \beta'$. This operation forgets still more information. It extracts the "abstract substitution part" of C. It is applied before executing a literal in a clause. See operation *EXTGS*.

$$\left.\begin{array}{l}\langle S, cf\rangle \in Cc(C),\\ \theta \text{ is an element of } S\end{array}\right\} \Rightarrow \theta \in Cc(\beta').$$

$EXT\text{-}NOCUT(\beta) = C'$. The empty prefix of the body of a clause produces a one element sequence with the information that no cut has been executed. This is expressed as follows:

$$\theta \in Cc(\beta) \Rightarrow \langle < \theta >, nocut\rangle \in Cc(C').$$

6 Abstract Semantics

The abstract semantics of a program P is defined as a set of abstract tuples (β, p, B) where p is a predicate symbol of arity n occurring in P, $\beta \in AS$, $B \in ASS$ and $dom(\beta) = dom(B) = \{x_1, \ldots, x_n\}$. The *underlying domain* UD of the program is the set of all (β, p) such that $\beta \in AS$, $dom(\beta) = \{x_1, \ldots, x_n\}$ and p occurs in P. In fact, we only consider sets of abstract tuples, *sat*, which are functions from UD into ASS and we use both $B = sat(\beta, p)$ or $(\beta, p, B) \in sat$. We denote $SATT$ the set of all those sets.

The abstract semantics is formally defined by means of a transformation $TSAT$ from $SATT$ to $SATT$.

Abstract Transformation The transformation $TSAT$ is in the same spirit as the transformation proposed in [9]. The main difference is that (output) abstract substitutions are replaced by abstract sequences. Abstract operations are modified accordingly. For example, the semantics in [9] uses an operation *UNION* to collect clause results. This operation is now replaced by operation *CONC*. Two major simplifications with respect to the concrete semantics have been however introduced to handle literals more simply. Let g be a goal of the form g', l and C be the abstract sequence resulting from the execution of g'. First, the input abstract sequence C for l is "abstracted" to a single abstract substitution β'' approximating all substitutions in the concretization of C, i.e. the sequence structure of C is "lost". Second, the input and ouput sequences for l are combined in all possible way through a unique operation *EXTGS*. This simplification was shown to provide a good trade-off between accuracy and efficiency in [2]. This trade-off could be reconsidered for more elaborate domains.

The abstract transformation $TSAT$ is defined in terms of the function T given in figure 1. T has arguments of the form $(\beta, cons, sat)$ where *cons* may be a predicate name p, a procedure or a suffix of a procedure (both denoted pr), a clause c or a goal g (i.e. the body or a prefix of the body of a clause). β is an abstract substitution whose domain agrees with the particular *cons*. *sat* is a set of abstract tuples. The result of T is either an abstract sequence B (for p and pr) or an abstract sequence with cut information C (for c and g).

$T(\beta, p, sat)$ executes $p(x_1, \ldots, x_n)$ with input abstract substitution β by calling the function $T(\beta, pr, sat)$ that executes all clauses defining p on β. $T(\beta, c.pr', sat)$ concatenates the results produced by the first clause c and by the rest of the procedure pr'. $T(\beta, c, sat)$ executes a clause by extending the abstract substitution β to all variables in c, executing the body and restricting the result to the variables in the head. $T(\beta, g, sat)$ executes the body of a clause by considering each literal in turn. The empty prefix of the body produces a one element abstract sequence with the information that no cut has been executed so far. When the next literal to execute is a cut, operation *AI-CUT* is executed. Otherwise the next literal l is executed with input β'' that approximates all substitutions in the concretization of C. Operation *RESTRG* expresses β'' in terms of the formal parameters $x_1, \ldots, x_n$ of l. If l is a procedure call then only a lookup in *sat* is performed, otherwise either operation *AI-VARS* or *AI-FUNCS* is executed. Operation *EXTGS* is performed after each call in order to obtain the result of the full goal.

Abstract Semantics Transformation $TSAT$ can be shown monotonic if the abstract operations are. However monotonicity is not an essential requirement for our framework because we do not define the abstract semantics as the least fixpoint of $TSAT$ which is not consistent in general as explained in section 2. In order to get a consistent *sat*, the transformation is applied to *sat*'s which are *pre-consistent* as defined below.

Definition 1 [pre-consistency] A set of abstract tuples *sat* is pre-consistent iff, for each abstract tuple $(\beta, p, B) \in sat$, $\langle \theta, p\rangle \longmapsto S$ with $\theta \in Cc(\beta)$ implies that there exists $S' \sqsubseteq S$ such that $S' \in Cc(B)$.

$$TSAT(sat) = \{(\beta, p, B) : (\beta, p) \in UD \text{ and } B = T(\beta, p, sat)\}$$

$T(\beta, p, sat) = T(\beta, pr, sat)$
where pr is the procedure defining p

$T(\beta, pr, sat) = SEQ(C)$
where $C = T(\beta, c, sat)$ if pr is c

$T(\beta, pr, sat) = CONC(\beta, C, B)$
where $B = T(\beta, pr', sat)$
$C = T(\beta, c, sat)$ if pr is $c.pr'$

$T(\beta, c, sat) = RESTRC(c, C)$
where $C = T(EXTC(c, \beta), g, sat)$
g is the body of c

$T(\beta, <>, sat) = C$
where $C = EXT\text{-}NOCUT(\beta)$

$T(\beta, (g, !), sat) = AI\text{-}CUT(C)$
where $C = T(\beta, g, sat)$

$T(\beta, (g, l), sat) = EXTGS(l, C, B)$
where $B =$ $AI\text{-}VARS(\beta')$ if l is $x_i = x_j$
$AI\_FUNCS(\beta', f)$ if l is $x_i = f(\ldots)$
$sat(\beta', p)$ if l is $p(\ldots)$
$\beta' = RESTRG(l, \beta'')$
$\beta'' = SUBST(C)$
$C = T(\beta, g, sat)$.

Figure 1: The Abstract Transformation

When there exists an abstract sequence $B_{<\perp>}$ such that $<\perp> \in Cc(B_{<\perp>})$, it is easy to define a first pre-consistent set of abstract tuples, since $<\perp> \sqsubseteq S$ for all S. Moreover, applying transformation $TSAT$ to pre-consistent *sats* gives other pre-consistent *sats* which are better lower approximations of the concrete outputs by consistency of the abstract operations. Finally, a postfixpoint is reached to obtain consistency. The abstract semantics can thus be formalized as any pre-consistent postfixpoint of the abstract transformation. Formally, the results can be stated as follows.

Lemma 2 Let *sat* be a pre-consistent set of abstract tuples. Then $TSAT(sat)$ is pre-consistent.

Theorem 3 [consistency of the abstract semantics] Let *sat* be a set of abstract tuples. If *sat* is a postfixpoint of $TSAT$, i.e. $TSAT(sat) \leq sat$, and is pre-consistent, then it is consistent. That is: for all $(\beta, p) \in UD$,

$$\left.\begin{array}{l}\langle\theta, p\rangle \longmapsto S, \\ \theta \in Cc(\beta)\end{array}\right\} \Rightarrow S \in Cc(sat(\beta, p)).$$

7 The Generic Abstract Interpretation Algorithm

We now discuss how postfixpoints of the abstract transformation can be computed. The key idea is that a postfixpoint can be computed by a generalization of existing generic abstract interpretation algorithms [3, 15, 9, 10]. We focus on the generalizations and their justifications here. See [2] for a description of the algorithm. The key generalization in the algorithm is the use of a more general form of widening, called E-widening, when updating the set of abstract tuples with a new result.

Definition 4 [E-widening] Let A be an abstract domain and B_i, B'_i be elements of A. A *E-widening* is an operation $\nabla : A \times A \to A$ which, given the sequences $B_1, \ldots, B_i, \ldots$ and $B'_0, \ldots, B'_i, \ldots$ such that $B'_{i+1} = B_{i+1} \nabla B'_i$ $(i \geq 0)$, satisfies

1. $B'_i \geq B_i$ $(i \geq 1)$;
2. there is a $j \geq 0$ such that all B'_i with $j \leq i$ are equal.

The E-widening is used as follows in the algorithm. Given an input pair (β, p), the output abstract sequence is computed by generating two sequences $B_1, \ldots, B_i, \ldots$ and $B'_0, \ldots, B'_i, \ldots$ as follows:

1. $B'_0 = B_{<\perp>}$ is stored in the initial *sat* as the output for (β, p);
2. B_i results from the *i-th* abstract execution of procedure p for abstract input β;
3. $B'_i = B_i \nabla B'_{i-1}$ is stored in the current *sat* after the *i-th* abstract execution of procedure p;
4. reexecution stops when $B_{i+1} \leq B'_i$.

Termination of the algorithm is guaranteed because all B'_i must be equal for all i greater than some j. Hence, since $B'_j = B'_{j+1}$ and $B'_{j+1} \geq B_{j+1}$, we have $B_{j+1} \leq B'_j$. Consistency of the result is guaranteed because each B'_i is pre-consistent and the algorithm terminates with a postfixpoint. Pre-consistency of the B'_i follows from $B'_i \geq B_i$ and the pre-consistency of B_i due to Lemma 2.

References

[1] M. Baudinet. Proving Termination Properties of Prolog Programs: A Semantic Approach. *Journal of Logic Programming*, 14, pages 1–29, 1992.

[2] C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality Analysis of Prolog. Technical report, Department of Computer Science, Brown University, March 1994. (Submitted to ILPS'94).

[3] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.

[4] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Type analysis of prolog using type graphs. In *Proceedings of PLDI'94*, Orlando, Florida, June 1994.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of (POPL'77)*.

[6] S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. Extracting Determinacy in Logic Programs. In *Proc. ICLP'93*, Budapest, June 1993.

[7] R. Giacobazzi. Detecting Determinate Computations by Bottom-up Abstract Interpretation. In *Proc. ESOP'92*, pages 167–181, 1992.

[8] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(4), 1992.

[9] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A generic abstract interpretation algorithm and its complexity analysis. In *Proceedings of (ICLP'91)*.

[10] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *(TOPLAS)*, January 1994.

[11] B. Le Charlier and S. Rossi. An Accurate Abstract Interpretation Framework for Prolog with cut (revised version). Technical report, Institute of Computer Science, University of Namur, Belgium. Forthcoming.

[12] J.W. Lloyd. *Foundations of Logic Programming.* Springer Series: Symbolic Computation–Artificial Intelligence. Springer-Velag, extended edition, 1987.

[13] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. In G. Ritter, editor, *Information Processing'89*, pages 601–606, San Fransisco, California, 1989.

[14] C.S. Mellish. Abstract interpretation of Prolog programs. In *Abstract Interpretation of Declarative Languages*, pages 181–198. Ellis Horwood Limited, 1987.

[15] K. Muthukumar and M. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2-3):315–347, August 1992.

[16] D. Sahlin. Determinacy Analysis for Full Prolog. In *Proc. PEPM'91*, 1991.

[17] D.A. Schmidt. *Denotational Semantics.* Allyn and Bacon, Inc., 1988.

[18] A. Taylor. LIPS on MIPS: Results from a Prolog Compiler for a RISC. In *Proceedings of (ICLP'90)*, Jerusalem, Israel, June 1990. MIT Press.

[19] P. Van Roy and A. Despain. High-Performance Computing with the Aquarius Compiler. *IEEE Computer*, 25(1), January 1992.

Semantics of concurrent logic programming as uniform proofs

Paolo Volpe
Dipartimento di Informatica, Università di Pisa
Corso Italia, 40 — 56125 Pisa, Italy
volpep@di.unipi.it

Abstract

We describe $\mathcal{LC}$, a formalism based on the proof theory of linear logic, whose aim is to specify concurrent computations and whose language restriction (as compared to other linear logic language) provides a simpler operational model that can lead to a more practical language core. The $\mathcal{LC}$ fragment is proved to be an abstract logic programming language, that is any sequent can be derived by uniform proofs. The resulting class of computations can be viewed in terms of multiset rewriting and is reminiscent of the computations arising in the Chemical Abstract Machine and in the Gamma model. The fragment makes it possible to give a logic foundation to existing extensions of Horn clause logic, such as Generalized Clauses, whose declarative semantics was based on an ad hoc construction. Programs and goals in $\mathcal{LC}$ can declaratively be characterized by a suitable instance of the phase semantics of linear logic. A declarative semantics, modeling answer substitutions, is associated to every $\mathcal{LC}$ program. Such a model gives a full characterization of the program computations and can be obtained through a fixpoint construction.

1 Introduction

The availability of powerful environments for parallel processing has made particularly interesting the field of logic languages. Writing concurrent programs is quite difficult. Therefore it is desirable to have languages with a clear and simple semantics, so as to have a rigorous basis for the specification, the analysis, the transformation and the verification of programs. A programming framework based on logic seems to be well suited.

In this paper we investigate the expressive power of linear logic in a concurrent programming framework. This logic is gaining wide consensus in theoretical computer science and our attempt is not quite new in its kind. Linear logic has already given the basis to many proposals. Our approach is based on the paradigm of computation as proof search, typical of logic programming. We take as foundation the proof theoretical characterization of logic programming given by Miller [26, 25]. The

definition of uniformity will lead us to single out a restricted fragment of linear logic capable of specifying an interesting class of parallel computations. We think the simple operational model can lead to a more practical language core.

The resulting framework is strongly related to the paradigm of multiset rewriting lying at the basis of the Gamma formalism [7] and of the Chemical Abstract Machine [9]. Actually it allows to specify a set of transformations that try to reduce an input multiset of goals to the empty multiset, returning as an output an answer substitution for the initial goal. More transformations can be applied concurrently to the multiset, thus making possible efficient implementations in parallel environments.

The rewriting of a multiset of logical formulas is amenable to simple interpretations in terms of process synchronization and communication, as shown in [14] and in [10]. We think that this provides a declarative notion of symmetrical communication and open the way to distributed programming.

In the last part of the paper we propose a semantics for the language obtained by instantiating the phase semantics of linear logic. The resulting abstract structure associated to programs allows to declaratively model the transformational behaviour of our computations. By exploiting the similarities of our fragment with the language of Generalized Clauses [14, 10], a declarative semantics modeling answer substitutions is also presented.

The paper is organized as follows. In subsection 1.1 we introduce linear logic and its proof system. In section 1.2 we introduce the definition of uniformity for multiple conclusions sequent systems. Section 3 shows the fragment $\mathcal{LC}$ and its computational features. In section 4 we relate the $\mathcal{LC}$ framework to actual programming environments. Finally a semantics for $\mathcal{LC}$ programs in the style of the phase semantics will be shown in section 5 together with a declarative semantics modeling answer substitutions and its fixpoint characterization.

1.1 Linear Logic

Linear logic is becoming an important subject the framework of computational logic. This is due to its interesting expressive features that make it possible to model both sequential and concurrent computations.

The key feature of the formalism is certainly its ability of treating resources through the manipulation of formulas, thus allowing to express in a natural way the notion of consumption and production. This leads to a direct interpretation of computation in linear logic. A process is also viewed as a (reusable) resource. The change of its state is obtained through the consumption (or decomposition) and the production (or construction) of resources. Since the processing of resources is inherently concurrent, multiple parallel flows of computation can be represented.

The sequent calculus allows us to represent in a natural way the resource sensitivity of this logic. A sequent can be thought of as an "action" that consumes the left-hand formulas and produces the right-hand ones. Equivalently a sequent tree makes explicit the consumption of formulas to produce more complex ones or vice versa the destruction of formulas in simpler parts.

The sequent system formalization shows also in a satisfactory way the subtle links between linear and classical logic. in fact we can see the derivation rules of the linear

$$\frac{}{\bot \vdash}(\bot L) \qquad \frac{}{\vdash 1}(1R) \qquad \frac{}{0, \Gamma \vdash \Delta}(0L) \qquad \frac{}{\Gamma \vdash \Delta, \top}(\top R)$$

$$\frac{}{A \vdash A}(id) \qquad \frac{\Gamma \vdash A, \Delta \quad A, \Lambda \vdash \Theta}{\Gamma, \Lambda \vdash \Delta, \Theta}(cut) \qquad \frac{\Gamma \vdash \Delta}{1, \Gamma \vdash \Delta}(1L) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \bot}(\bot R)$$

$$\frac{A, B, \Gamma \vdash \Delta}{A \otimes B, \Gamma \vdash \Delta}(\otimes L) \qquad \frac{\Gamma \vdash \Delta, A \quad \Lambda \vdash \Theta, B}{\Gamma, \Lambda \vdash \Delta, \Theta, A \otimes B}(\otimes R) \qquad \frac{A, \Gamma \vdash \Delta \quad B, \Lambda \vdash \Theta}{A \parr B, \Gamma, \Lambda \vdash \Delta, \Theta}(\parr L) \qquad \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \parr B}(\parr R)$$

$$\frac{\Gamma \vdash \Delta, A \quad B, \Lambda \vdash \Theta}{A \multimap B, \Gamma, \Lambda \vdash \Delta, \Theta}(\multimap L) \qquad \frac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \multimap B}(\multimap R) \qquad \frac{\Gamma \vdash \Delta, A}{A^{\bot}, \Gamma \vdash \Delta}(\bot L) \qquad \frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, A^{\bot}}(\bot R)$$

$$\frac{A, \Gamma \vdash \Delta}{A \& B, \Gamma \vdash \Delta}(\&L) \qquad \frac{B, \Gamma \vdash \Delta}{A \& B, \Gamma \vdash \Delta}(\&L) \qquad \frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \& B}(\&R)$$

$$\frac{A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta}{A \oplus B, \Gamma \vdash \Delta}(\oplus L) \qquad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \oplus B}(\oplus R) \qquad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \oplus B}(\oplus R)$$

$$\frac{A, \Gamma \vdash \Delta}{!A, \Gamma \vdash \Delta}(!L) \qquad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, ?A}(?R) \qquad \frac{\Gamma \vdash \Delta}{!A, \Gamma \vdash \Delta}(!WL) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, ?A}(?WR)$$

$$\frac{!\Gamma, A \vdash ?\Delta}{!\Gamma, ?A \vdash ?\Delta}(?L) \qquad \frac{!\Gamma \vdash A, ?\Delta}{!\Gamma \vdash !A, ?\Delta}(!R) \qquad \frac{!A, !A, \Gamma \vdash \Delta}{!A, \Gamma \vdash \Delta}(!CL) \qquad \frac{\Gamma \vdash \Delta, ?A, ?A}{\Gamma \vdash \Delta, ?A}(?CR)$$

$$\frac{A[t/x], \Gamma \vdash \Delta}{\forall x A, \Gamma \vdash \Delta}(\forall L) \qquad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, \forall x A}(\forall R) \qquad \frac{A, \Gamma \vdash \Delta}{\exists x A, \Gamma \vdash \Delta}(\exists L) \qquad \frac{\Gamma \vdash \Delta, A[t/x]}{\Gamma \vdash \Delta, \exists A}(\exists R)$$

Table 1: The sequent system of linear logic

logic sequent system obtained from the classical derivation rules by abolishing the contraction and the thinning rules, which are responsible for the insensitivity of classical logic to formulas multiplicity. This elimination causes the classical connectives and constants to split down into two versions, the additive and the multiplicative one. In order to reach the power of classical logic, two modalities have to be introduced in order to have formulas *reusable* ad libitum. Contraction and thinning rules are recovered for formulas annotated with the modalities.

In Table 1 we show the sequent system $\mathcal{LL}$ of linear logic. A sequent is an expression $\Gamma \vdash \Delta$, where Γ and Δ are multisets of linear formulas, i.e. formulas built on the linear primitives $\otimes, \parr, \oplus, \&, 1, 0, \bot, \top, \forall, \exists, !, ?$. Notice the use of the multiset structure, which allows us to get rid of an explicit structural rule of exchange. For a more complete account on the $\mathcal{LL}$ system we refer to [16].

The outstanding features of linear logic have soon determined its impact on theoretical computer science. Actually two basic approaches can be recognized. The *functional approach* models computations through normalization, i.e. *cut elimination*, of sequent calculi derivation trees. This approach has allowed the definition of powerful functional languages where we can express concurrent computations and derive program properties (e.g. strictness, sharing), which make it possible the elimination of the garbage collector [1, 21, 22].

We are more concerned with the other approach, the *logic programming approach*. In the following we will illustrate in details how we have instantiated this paradigm. Let us just note that the "proof search as computation" analogy applied to linear logic has already inspired several logic programming frameworks like *LinLog* [2], *Linear Objects* [3, 4], and Miller's linear refinement of hereditary Harrop's formulas [18].

1.2 Modeling computations in a sequent system

As already mentioned our approach is based on the notion of computation as proof search. In particular the proofs we search are cut-free derivations in a sequent calculus. As usual a derivation statically represents the evolution of a computation. We want a sequent $\Gamma \vdash G_1, \ldots, G_n$ in that derivation to represent the evolution of a computation. The change of state through the derivation tree is obtained by applying derivation rules. The right side of a sequent is viewed as a multiset of agent formulas that evolves through applications of derivation rules. Each complex agent can be decomposed in a uniform way and independently from the other formulas of the sequent. The evolution of atomic agents, possibly together with other atoms, is instead dependent on other formulas of the sequent.
The left side of the sequent is viewed as a set of rules which specifies how simple right formulas (alone or in a group) can be transformed.

In other words we want our proofs to satisfy a suitable "parallel" extension of the notion of *uniformity* [26]. We remember that in [26] the definition of uniformity was given for single conclusion sequent derivations only, so it does not seem suitable to our aim. Miller in [25] has proposed an extension of that definition for multiple conclusions sequent derivations similar to ours. This definition formally singles out exactly the class of derivations we are concerned with.

Definition 1
A sequent proof Ξ is uniform if it is cut-free and for every subproof Ψ of Ξ and for every non atomic formula occurrence B, in the right-hand side of the root sequent of Ψ, there exists a derivation Ψ' which is equal to Ψ up to permutation of inference rules and such that the last inference rule in Ψ' introduces the top level logical connective occurring in B. ∎

It can easily be shown that this definition generalizes the one given in [26]. This definition formalizes a "concurrent" view of derivations where the permutability represents the ideal simultaneous application of several independent right introduction rules. If two or more introduction rules can be applied, all derivations using them can be obtained from each other by simply permuting the order of rules application.

As mentioned in [26], restricting the proof search we implicitly give a fixed operational meaning to the logical primitives. We can thus define a basic formalism which can specify concurrent computations and which is very reminiscent of models like CHAM and Gamma, based on multiset rewriting.

Moreover, since we are working in a logical system we have a declarative interpretation for agent-formulas and a notion of logical equivalence and we can give an abstract semantics to computations based on the declarative semantics of the logic we use. However we must be consistent with the declarative meaning of the logical primitives. Thus our task can be stated as follow: Given a logical system we want to single out a subset of the well formed sequents such that if they are derivable they can be derived uniformly. This guarantees that the operational meaning of the logical primitives be consistent with their declarative meaning. In other words, even if we search uniform proofs only we do not lose the completeness.

Fragments enjoying this property are called by Miller *abstract logic programming languages*. Therefore our aim can be stated as the definition of an abstract logic programming language in linear logic, able to model concurrent computations. As we will see, linear logic has such constructive features to allow that.

2 Preliminaries

We assume the reader to be familiar with the terminology and the basic results of the semantics of logic programs [5]. The language alphabet is $\langle D, V, P \rangle$, with D a finite set of function symbols, V a denumerable set of variables, P a finite set of predicate symbols. The set $T_D(V)$ of *terms* is defined as follows: i) $\forall c \in D$, c constant, $c \in T_D(V)$; ii) $\forall x \in V$, $x \in T_D(V)$; iii) $\forall t_1, \ldots, t_n \in T_D(V)$ and $\forall f \in D$ of arity n, $f(t_1, \ldots, t_n) \in T_D(V)$. $T_D(\emptyset)$ is the set of *closed* terms. A *substitution* is a mapping $\theta : V \to T_D(V)$ such that $dom(\theta) = \{X \mid \theta X \neq X\}$ is finite. A substitution is denoted with the expressions $\{X_1/t_1, \ldots, X_n/t_n\}$ or $[X_1/t_1, \ldots, X_n/t_n]$ specifying that the variables $X_1, \ldots, X_n$ are mapped to $t_1, \ldots, t_n$. The symbol ϵ denotes the empty substitution. The composition $\theta\sigma$ of the substitutions θ and σ is defined as the functional composition. The pre-ordering $\leq$ on substitutions is such that $\theta \leq \sigma$ iff $\exists \rho$ such that $\theta = \sigma\rho$. An *atom* is an object $p(t_1, \ldots, t_n)$ with $p \in P$ and $t_1, \ldots, t_n \in T_D(V)$. A is *closed* or *ground* if all of its terms belongs to $T_D(\emptyset)$. A *linear formula* is a well formed formula built on atoms and on the linear primitives $\otimes$, $\parr$, $\oplus$, $\&$, 1, 0, $\bot$, $\top$, $\forall$, $\exists$, $!$,$?$. A *linear sequent* is an expression $\Gamma \vdash \Delta$ with Γ and Δ multisets of linear formulas. A linear formula is closed if it has not free variables. $\mathcal{M}_{LL}$ denotes the set of multisets of linear closed formulas. Given the expressions A and A', we define $A \leq A'$ iff $\exists \theta$ such that $A = A'\theta$ ($A'\theta$ is the result of the application of the θ to the variables of A'). Let $\approx$ be the associated equivalence relation (*renaming*). Two expressions A_1 and A_2 are *unifiable* is exists a substitution θ, a *unifier*, such that $A_1\theta = A_2\theta$. The *most general unifier* of two expressions is the maximal unifier (w.r.t. $\leq$). The notation $\bar{X}$ denotes a tuple of different variables. An atom is *maximal* if it is of the form $p(\bar{X})$. A *maximal multiset* is a multiset of maximal atoms. Finally $var(E)$ denotes the set of the free variable of the expression E.

3 The $\mathcal{LC}$ fragment

In this section we single out a fragment of linear logic that seems to be adequated for specifying an interesting class of computations. We will define a subset of the linear sequents by stepwise approximation, justifying every extension in terms of an improvement of the expressive power. Eventually we will identify a class of uniform proofs for the sequents of the subset. Our aim will be then to show that when considering uniform proofs only we keep the declarative meaning of the logical primitives. This will be done by proving that the fragment is an abstract logic programming language. Let us remind that we want to represent concurrent computations as the concurrent evolution of the agents $G_1, \ldots, G_n$ in the proof tree of the sequent $\Gamma \vdash G_1, \ldots, G_n$, where Γ contains rules for the evolution of the agents. As a first step we can assume

the agents to be simple ground atoms.

First of all we want to have "transformation formulas", like $G \rightarrow A$, that allow a change of state in the evolution of the atomic agent A. The application of this formula transforms the atom A into the agent G. We can easily obtain this behaviour by using linear formulas like $G \multimap A$, where $\multimap$ is linear implication. The above operational meaning can be assigned to those clauses by specifying how a proof search can evolve, once we have an atom A in the multiset and the formula $G \multimap A$ in the right-hand side of the sequent.

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash G, \Delta\end{array} \quad \dfrac{}{A \vdash A}(id)}{\Gamma, G \multimap A \vdash A, \Delta}(\multimap L)$$

This operational meaning is fixed, by forcing the rule $(\multimap L)$ to be applied only that way. Notice that, due to the loss of the contraction rule (we cannot duplicate formulas), the formula A of the conclusion sequent is no more present in the continuation of the computation (the subproof starting from $\Gamma \vdash G, \Delta$). In fact the loss of this rule creates a problem: as it can easily be noticed, once applied, the formula $G \multimap A$ is no more applicable in the rest of the computation. Since we want these clauses to be usable more than once, we can mark all of them with the modality !, thus making them reusable resources of the computation. Through a clever use of the rules $(!C)$ and $(!D)$ we can specify the multiple reuse of "program" formulas. The following is an example of the resulting behaviour.

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ !\Gamma, !(G \multimap A) \vdash G, \Delta\end{array} \quad \dfrac{}{A \vdash A}(id)}{!\Gamma, !(G \multimap A), G \multimap A \vdash A, \Delta}(\multimap L)}{!\Gamma, !(G \multimap A), !(G \multimap A) \vdash A, \Delta}(!D)}{!\Gamma, !(G \multimap A) \vdash A, \Delta}(!C)$$

Thus the computation evolves without consuming the clause $!(G \multimap A)$. Due to the constructive capabilities of linear logic an interesting change can be made to program formulas. The application of a clause can easily be extended to involve more than one atom, that is more than one agent. By considering formulas like $G \multimap A \parr B$ we can obtain the following transition

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, !(G \multimap A \parr B) \vdash G, \Delta\end{array} \quad \dfrac{\dfrac{}{A \vdash A}(id) \quad \dfrac{}{B \vdash B}(id)}{A \parr B \vdash A, B}(\parr L)}{!\Gamma, !(G \multimap A \parr B), G \multimap A \parr B \vdash A, B, \Delta}(\multimap L)}{!\Gamma, !(G \multimap A \parr B), !(G \multimap A \parr B) \vdash A, B, \Delta}(!D)}{!\Gamma, !(G \multimap A \parr B) \vdash A, B, \Delta}(!C)$$

Obviously the clauses can have more than two atoms in their heads. The application of these "multiset rewriting rules" can be interpreted in a variety of ways. [14] and [10] suggest to view them as a way to synchronize n computations and to allow

them to communicate through a full not constrained unification. [10] also suggests to view them as specifying the consumption of "atomic messages", floating in the multiset, considered as a pool of processes and messages. An asynchronous model of communication is thus achieved. Some example in the following section will show the different forms of communication.

Clauses need not to be ground. It is a straightforward task to lift them to first order clauses by obtaining schemas of transformations. Clauses have then the form $\forall \vec{x}(G \multimap A_1 \parr \cdots \parr A_n)$, with the free variables of G included in the free variables of $A_1 \parr \cdots \parr A_n$. Obviously the rule $(\forall L)$ will be used in derivations. The structure of goal formulas can be made more complex, so as to be decomposed in the derivation through the application of right introduction rules.

First of all we can use the connective $\parr$ to connect more elementary goal formulas. Consider the $(\parr R)$ introduction rule.

$$\dfrac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \parr B, \Delta}(\parr R)$$

We can view the previous behaviour as the decomposition of the agent $A \parr B$ into the two agents A and B. In other terms we can specify the creation of independent flows of subcomputation inside the overall computation.

Another connective that can be added to our fragment is the connective $\oplus$. The right introduction rules for $\oplus$ are the following:

$$\dfrac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \oplus B, \Delta}(\oplus R) \qquad \dfrac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \oplus B, \Delta}(\oplus R)$$

We can have then an operator to specify internal nondeterminism. A possible extension can be obtained by introducing guards.

We can also have existentially quantified variables in the agents. This is the basis for a notion of computed substitution as output of the computation (the substitutions will obviously be the bindings of the existentially quantified variables of the starting agents multiset).

Other behaviours can be specified by using the constants of linear logic. In particular we can easily force the termination of a computation or the disappearing of an agent in a multiset. We can use the introduction rule for the constant $\top$ to specify the termination of the overall computation like in the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{!\Gamma, !(\top \multimap A) \vdash \top, B, C}(\top R) \quad \dfrac{}{A \vdash A}(id)}{!\Gamma, !(\top \multimap A), \top \multimap A \vdash A, B, C}(\multimap L)}{!\Gamma, !(\top \multimap A), !(\top \multimap A) \vdash A, B, C}(!D)}{!\Gamma, !(\top \multimap A) \vdash A, B, C}(!C)$$

The application of the transformation $\top \multimap A$ ends the computation. We can stop the proof search since we have obtained a correct proof of the final sequent.

The other feature, the disappearing of an agent in the multiset, can be obtained by using the constants 1 and $\bot$. Actually we can use the clauses $1 \multimap A$ and $\bot \multimap A$ to state

that the agent A can disappear from a multiset. The clause $\bot\multimap A$ makes the agent A disappear in a non empty multiset (see rule $(\bot R)$). The clause $1\multimap A$ makes the agent A end (together the overall computation) in the empty agent multiset (see rule $(1R)$). We will use for the pair of clauses $!(\bot\multimap A), !(1\multimap A)$ the equivalent notation $!(1 \oplus \bot\multimap A)$. Two termination modes (stated in terms of communicating processes) are then made available; i.e. silent termination by using the formula $1 \oplus \bot$ and overall termination by using the constant $\top$.

Thus we have singled out an interesting fragment of linear logic with a class of uniform derivations (as it can easily be proved) which can express computations in the style of multiset rewriting. In our derivations the clause formulas act as multiset rewriting rules, while the agent formulas are uniformly decomposed in the derivation, until they are reduced to atomic formulas which can then be rewritten.
We do not want to lose the declarative reading of the logical primitives we use. As we will show in the following this is not the case since our fragment is an abstract logic programming language, that is the search of uniform derivations restricted to the fragment is complete.

In order to state this result we summarize in a more formal way the properties of our fragment. The fragment of linear logic (we will call it $\mathcal{LC}$ for *Linear Chemistry*) we are considerating is composed of the sequent $!\Gamma \vdash \Delta$, where Γ and Δ are finite multisets of formulas of the sets $\mathcal{D}$ and $\mathcal{G}$, respectively, of linear logic formulas. The set $\mathcal{G}$ is composed by the goal formulas G defined as

$$G := A | 1 \oplus \bot | \top | G \parr G | G \oplus G | \exists x G$$

where A is an atomic formula and G is a goal of $\mathcal{G}$. The set $\mathcal{D}$ is composed by the clause formulas D defined as

$$D = \forall (G \multimap A_1 \parr \cdots \parr A_n)$$

where $A_1, \ldots, A_n$ are atomic formulas and G is a formula of $\mathcal{G}$. The free variables of G are included in the free variables of $A_1, \ldots, A_n$.

The following theorem asserts the completeness of a uniform proof search strategy for the sequent of $\mathcal{LC}$.

Theorem 1 *Let Γ be a multiset of formulas of $\mathcal{D}$ and Δ be a multiset of formulas of $\mathcal{G}$. Then $!\Gamma \vdash \Delta$ is derivable in linear logic if and only if $!\Gamma \vdash \Delta$ is uniformly derivable.*

The fragment $\mathcal{LC}$ is then an abstract logic programming language. On the basis of theorem 1 we can state that the derivation rule system in Table 2 (we will call it $\mathcal{LC}$ system) is sound and complete w.r.t. linear logic for the sequents of $\mathcal{LC}$. In the rule $(\multimap)$ the formula $G\multimap A_1 \parr \cdots \parr A_n$ is an instance of $G'\multimap A'_1 \parr \cdots \parr A'_n$ and $!\forall(G'\multimap A'_1 \parr \cdots \parr A'_n)$ is in Γ. In the rule $(\exists)$ the formula $A[t/x]$ is obtained by substituting all the free occurrences of x in A by a term t.

These rules explicitly convey the computational flavour of the primitives of the language. It can easily be shown that they represent in a compact form exactly the uniform derivations considered introducing the $\mathcal{LC}$ formulas. A sequent $\Gamma \vdash \Delta$ will be $\mathcal{LC}$ derivable ($\Gamma \vdash_{\mathcal{LC}} \Delta$) if it can be derived by the $\mathcal{LC}$ derivation rules.

$$\frac{}{!\Gamma \vdash 1 \oplus \bot}(1 \oplus \bot) \qquad \frac{}{\Gamma \vdash \top, \Delta}(\top) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash 1 \oplus \bot, \Delta}(1 \oplus \bot)$$

$$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \parr B, \Delta}(\parr) \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \oplus B, \Delta}(\oplus) \qquad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \oplus B, \Delta}(\oplus)$$

$$\frac{\Gamma \vdash G, \Delta}{\Gamma \vdash A_1, \ldots, A_n, \Delta}(\multimap) \qquad \frac{\Gamma \vdash A[t/x], \Delta}{\Gamma \vdash \exists x A, \Delta}(\exists)$$

Table 2: The system $\mathcal{LC}$.

An important *observable* of $\mathcal{LC}$ computations is certainly the answer substitutions, that is the substitution found by the proof for the existentially quantified variables in the initial goals. In fact the answer substitution is commonly considered as the output of a logic language program execution. We think then that a semantics able to model correctly the answer substitutions of a program could be certainly very useful for applications such as program transformation, program verification, program analysis. With this aim in view we will give a particular relevance to the "answer substitution behaviour" of a derivation introducing the following notions.

Definition 2
We will write $P \vdash_{\mathcal{LC}} \Delta$ *by* Π *with substitution* θ to say that the existentially quantified variables in Δ are instantiated with the substitution θ in the derivation Π. ∎

From this definition we can single out the following equivalence between $\mathcal{LC}$ programs.

Definition 3
Two programs P_1 and P_2 are *answer substitution equivalent* if for each multiset Δ and for each substitution θ, $P_1 \vdash_{\mathcal{LC}} \Delta$ with substitution θ iff $P_2 \vdash_{\mathcal{LC}} \Delta$ with substitution θ. ∎

Anyway this equivalence results to be too abstract. We think that a good equivalence criterium should be based instead on a notion of "most general answer substitution", modeling the substitution restituted by a proof procedure based on unification. In fact unification is the best choice in an implementation to reduce the non-determinism of proof rules such as $(\exists)$. A semantics capable of modeling the answer most general substitution has revealed in ([8]) to be very adequate for data-flow analysis problems of traditional logic languages, employing efficient proof procedure, such as SLD resolution, based on unification. We introduce then the following definitions to express a notion of "unification answer substitution" inside the system $\mathcal{LC}$.

Definition 4
The derivation Π is *more general* than the derivation Π' if there exists a substitution θ such that $\Pi' = \Pi\theta$, i.e. Π' is obtained from Π instantiating its free variables. ∎

Definition 5
A derivation Π is a *most general derivation (mgd)* if for each derivation Π', $\Pi = \Pi'\theta$ implies $\exists\sigma$ such that $\Pi' = \Pi\sigma$, that is Π is maximal w.r.t. the "more general" relation. ∎

Definition 6
With $P \vdash_{\mathcal{LC}} \Delta$ *with most general substitution (mgs)* θ we mean that exists a most general derivation Π of $P \vdash \Delta$ and $P \vdash_{\mathcal{LC}} \Delta$ by Π with substitution θ. ∎

Definition 7
Two programs P_1 and P_2 are *mgs equivalent* if for each multiset Δ and for each substitution θ $P_1 \vdash_{\mathcal{LC}} \Delta$ with mgs θ iff $P_2 \vdash_{\mathcal{LC}} \Delta$ with mgs θ. ∎

We can say then that if $P \vdash_{\mathcal{LC}} \Delta$ with mgs θ then a unification based proof procedure will find the substitution θ for the sequent $P \vdash \Delta$. For a given sequent there can exist, in general, several most general derivations. It can be easily proved that once fixed a sequence of program clauses to be used in a derivation, again multiple mgd's can be found (differing for inessential permutations in the order of the rules) but all have the same mgs.

In the next sections we will see a semantics that model correctly mgs equivalence (indeed it is fully abstract), based on the S-semantics approach [13].

4 $\mathcal{LC}$ and the logic Programming

We think that $\mathcal{LC}$ provides interesting features as a programming language. First of all it can be shown that we can embed in $\mathcal{LC}$ the language of Generalized Horn Clauses ($\mathcal{GC}$), introduced in [23, 24, 27] and further developed in [14] and [10]. $\mathcal{GC}$ allows multiple atoms in the heads of clauses with the aim of synchronizing concurrent computations. The syntax of the clauses is the following

$$A_1 + \cdots + A_n \leftarrow B_1 + \cdots + B_m \quad n \geq 1, m \geq 0$$

The operational semantics is a straightforward generalization of SLD refutation. $\mathcal{GC}$ formulas can easily be translated to formulas of $\mathcal{LC}$ thus preserving the computational behaviour. The translation is the following.

$$\begin{aligned} (A)^\circ &= A \text{ if A is an atom;} \\ (\Box)^\circ &= 1 \oplus \bot; \\ (A + B)^\circ &= (A)^\circ \parr (B)^\circ; \\ (A \leftarrow B)^\circ &= \forall(\exists(B)^\circ \multimap (A)^\circ). \end{aligned}$$

The $\exists$ quantifier binds all the variables of $(B)^\circ$ not included in $(A)^\circ$. The $\forall$ quantifier binds all the free variables of $\exists(B)^\circ \multimap (A)^\circ$. The translation of $\Box$ to $1 \oplus \bot$ is explained by noting that in $\mathcal{GC}$ AND-parallel processes die silently without disturbing the others and that can be obtained through the disappearing of a $\mathcal{LC}$ agent in a multiset. It is not hard to prove the following theorem that relates $\mathcal{GC}$ to $\mathcal{LC}$.

Theorem 2 *Let P be a $\mathcal{GC}$ program and $A_1, \ldots, A_n$ be atomic formulas. The goal $\leftarrow A_1 + \cdots + A_n$ is refutable in P with computed answer substitution θ iff $!\Gamma \vdash ?(A_1 \parr \cdots \parr A_n)$ is $\mathcal{LC}$-provable with mgs θ, where Γ is the multiset of the translations of the clauses of program P and $\exists$ binds all the variables in $A_1 \parr \cdots \parr A_n$.*

We can now show that $\mathcal{LC}$ can face the same class of problems for which $\mathcal{GC}$ was introduced. In the following we will show some programming examples taken from [14] and [10] to show in practical cases the kind of behaviours $\mathcal{LC}$ can specify.

The application of multiple head clauses can be viewed as a synchronization mechanism between agents. Moreover, if we assume the use of unification in the proof search, this application makes it possible the symmetrical exchange of messages.

Example 4.1 Let us consider the program $\mathcal{P}$

$$\begin{aligned} & SRB(v,X) \parr bodyA(X) \multimap A \\ & SRA(Y,w) \parr bodyB(Y) \multimap B \\ (*) \quad & 1 \oplus \bot \multimap SRB(X,Y) \parr SRA(X,Y) \end{aligned}$$

The computation starts with $\mathcal{P} \vdash A, B$. We have two processes communicating through the application of the program clause $(*)$. This clause will consume in the multiset the two atoms $SRB(v,X)$ and $SRA(Y,w)$ and at the same time binds Y to v and X to w, thus allowing an exchange of information between the two processes. ∎

This communication model can easily be extended to allow multiple agents to exchange information.

Example 4.2 The clause

$$p_1(X_1, \ldots, X_k) \parr \cdots \parr p_k(X_1, \ldots, X_k) \multimap p_1(X_1, \ldots, X_k) \parr \cdots \parr p_k(X_1, \ldots, X_k)$$

when applied in an environment which contains the atoms

$$\ldots, p_1(v_1, Y_1^2, \ldots, Y_1^k), \ldots, p_i(Y_i^1, \ldots, v_i, \ldots, Y_i^k), \ldots, p_k(Y_k^1, \ldots, Y_k^{k-1}, v_k), \ldots$$

causes every agent p_i to communicate the value v_i to the other processes and to receive from them $k-1$ messages (we are assuming variables in the agents to be *logical* variables). Obviously the clause has to be opportunely instantiated to preclude it from keeping on reapplying itself. ∎

As already noted, another model of communication can easily be obtained. We can distinguish two types of objects in the right-hand of a sequent. Besides the agent formulas which can be viewed as process activations, we can single out some atomic formulas acting as messages. The application of a multiple head formula can then be seen as the consumption of some message and possibly as the production of new ones. An asynchronous communication paradigm can then be established. This view of $\mathcal{LC}$ computations can improve the modularity of the program design.

Example 4.3 The following program, taken from [28], defines two processes which cooperate to build a list of squares exploiting the fact that the sum of the first n odd numbers equals n^2:

$o(N,s(0)) \multimap odd(N)$
$end \multimap o(0,I) \parr ok$
$o(N,s(s(I))) \parr num(I) \multimap o(s(N),I) \parr ok$

$ok \parr q(0,K) \multimap sqr(K)$
$1 \oplus \bot \multimap q(Q,Q.nil) \parr end$
$ok \parr \exists R(add(J,Q,R) \parr q(R,K)) \multimap q(Q,Q.K) \parr num(J)$

The computation originated from the initial goal formulas $odd(3), \exists K\, sqr(K)$, binds the variable K to the list $0.1.4.9.nil$. The *odd* process computes the first 3 odd numbers sending to the environment a *num* message for each one of them. The *sqr* process consumes the *num* message, adds the numbers and stops as soon as it receives the *end* message. The *ok* message has been introduced to synchronize the process. ∎

The resulting communication model is reminiscent of Linda's *generative communication*[15]. Finally, as noted in [10], these two schemes of communication can be used in such a way as to make useless the sharing of variables between different agents. We can easily impose that agents do not share variables without losing any expressive power. This makes it possible the use of $\mathcal{LC}$ in a distributed environment.

5 The semantics

Since our fragment is an abstract logic programming language we can be confident about the use of the semantics of linear logic to characterize the $\mathcal{LC}$ computations. An interesting semantics for $\mathcal{LC}$ programs and goals is obtained by instantiating the phase semantics of linear logic, the semantics proposed by Girard in his seminal paper [16], to which we refer for a complete account.

In the following we consider a *phase semantics* simply as a triple $(M, \bot, s)$, where M is a monoid, $\bot$ is a subset of M, s is a valuation of atomic formulas in M, i.e. a function mapping atoms into specific subsets of M (the *facts*). By structural induction on the linear formulas and by using the distinguished operators $\parr_M, \oplus_M, \ldots$ on the facts of M, the function s can be extended to the whole set of well formed formulas (for example $s(A \parr B) = s(A) \parr_M s(B)$). Finally a formula A is considered valid if it is the case that $1 \in s(A)$.

Given a program P, we obtain a phase semantics $M_P = (\mathcal{M}_{LL}, \bot_P, s)$ which behaves like the canonical Herbrand models of traditional logic programs. Namely the validity of a goal formula in the model amounts to its provability from the program P. Let us define now in detail the *canonical phase model* of the program P. We associate to a program P the phase model $M_P = (\mathcal{M}_{LL}, \bot_P, s)$, on the base $\mathcal{M}_{LL}$ of closed multisets, obtained as follows:

- $\bot_P = \{\Delta | \Delta \in \mathcal{M}_{LL}$ and $P \vdash_{LC} \Delta, 1 \oplus \bot\}$,
- $s(A) = \{\Delta | \Delta \in \mathcal{M}_{LL}$ and $P \vdash_{LC} A, \Delta\ \}$.

In other words $\bot_P$ is the subset of $\mathcal{M}_{LL}$ composed by all the multisets which start a successful computation. Finally every atom A is mapped by s into the subset of

$\mathcal{M}_{LL}$ composed of the complementary environment of A in a successful computation. It can easily be verified that M_P is a phase semantics.

The phase semantics M_P singles out in a standard way the operators between facts $\parr_M, \oplus_M, \multimap_M$, the function $!_M$ and the distinguished facts $\mathbf{I}, \bot_P, \top$, that allow to extend the valuation s to map every formula of the $\mathcal{LC}$ fragment into a fact of the model.

Lemma 1 *Given a program P and its phase semantics M_P then for each $G \in \mathcal{G}$*

$$s_{\text{ext}}(G) = \{\Delta \in \mathcal{M}_{LL} | P \vdash G, \Delta\}$$

where s_{ext} is the extension of s to linear formulas.

Theorem 3 *$P \vdash G$ is $\mathcal{LC}$-provable if and only if G is valid in M_P.*

We have thus obtained a class of models for the language $\mathcal{LC}$ which generalizes the Herbrand models of traditional logic programming languages [5]. While the latter can be viewed as mapping from classical goal formulas into the set {true, false}, our model maps a goal formula into a fact of M_P. Note that in the case of a single head clauses program P, not using the constant $\top$, the phase model semantics M_P reduces to a boolean evaluation for the closed goals of $\mathcal{LC}$.

Example 5.1 We build the phase model of the program shown in example 4.3. The base $\mathcal{M}_{LL}$ is the set of multisets of closed goals in the language $\{\{odd/1, o/2, end/0, num/1, sqr/1, q/2, add/3\}, \{0, s/1\}\}$.

$$\bot_P = \{\ [\top],\ [odd(0), sqr(0.nil)],\ [odd(s(0)), sqr(0.s(0).nil)],\ \ [1 \oplus \bot],\ [o(0, s(0)), sqr(0.nil)],\ [o(s(0), s(0)), sqr(0.s(0).nil)],\ \ [\top, 1 \oplus \bot],\ [o(0, s(0)), q(0, 0.nil)],\ [o(s(0), s(0)), q(0, 0.s(0).nil)],\ \ \ldots\ \ldots\ \ldots\}$$

The interpretation of $odd(0)$, for example, can easily be extracted from $\bot_P$.

$$s(odd(0)) = \{\ \begin{array}{lll} [sqr(0.nil)], & [q(0, 0.nil)], & \ldots \\ [sqr(0.nil), 1 \oplus \bot], & [q(0, 0.nil), 1 \oplus \bot] & \ldots \\ [sqr(s^3(0)), \top], & [q(s^3(0), \top] & \ldots\} \end{array}$$

∎

Anyway this semantics is not very satisfactory. It does properly model only the success behaviour of the goal formulas. It fails to model correctly the most general answer substitutions of programs. For the reasons previously expressed we think it would be more useful a semantics less abstract, able to model that more concrete observable. In the following then, on the base of the S-semantics approach to the semantics of logic programs proposed by Falaschi et al. [13], we propose a S-semantics that captures the most general substitution equivalence among programs, giving an operational and a fixpoint characterization of it.

5.1 Operational characterization of S-semantics

Like in [13], the semantics we propose is a set, composed of multisets of non-ground atoms. To the aim of defining it we introduce the non-ground multiset base $\mathcal{M}_V$, whose subsets will be the semantics of our programs.

Definition 8
The non ground multiset base $\mathcal{M}_V$ is the set of all the multisets of atoms $p(t_1, \ldots, t_n)$, with $p \in P$, p of arity n and $t_1, \ldots, t_n \in T_D(V)_{/\approx}$, where $\approx$ is the variance equivalence relation. ∎

The S-semantics of the $\mathcal{LC}$ program P is obtained as a result of the following operational (top-down) construction

$$O(P) = \{\Delta | P \vdash_{LC} \Delta \text{ with mgs } \theta, \Delta = \exists p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)\}$$

$O(P)$ is a subset of $\mathcal{M}_V$. The following theorems show that O models correctly most general answer substitutions and that it is *fully abstract.*

Theorem 4 *$P \vdash_{LC} \Delta$ with mgs θ iff $O(P) \vdash_{LC} \Delta$ with mgs θ.*

We obtain as a simply corollary that O is fully abstract.

Theorem 5 *P_1 and P_2 are mgs equivalent iff $O(P_1) = O(P_2)$.*

5.2 Fixpoint characterization of S-semantics

An extended immediate consequence operator T_P on subsets of M_V can be introduced, whose least fixpoint will be shown to be equivalent to the most general answer substitution semantics $O(P)$.

Lemma 2 *$(\mathcal{P}(\mathcal{M}_V), \subseteq)$ is a complete lattice.*

For the definition of T_P we need the following relation.

Definition 9
Let G be a goal formula and I a subset of $\mathcal{M}_V$. $G \ll I$ is true if

- $G = A_1 \parr \cdots \parr A_n$, $A_1, \ldots, A_n$ atoms and $[A_1, \ldots, A_n]_{/\approx} \in I$;
- $G = 1 \oplus \bot$ or $G = \top$;
- $G = \top \parr H$, $H \in \mathcal{M}_V$;
- $G = G_1 \parr G_2$ and $G_1 \ll I$ and $G_2 \ll I$;
- $G = G_1 \oplus G_2$ and ($G_1 \ll I$ or $G_2 \ll I$);
- $G = G_1 \parr (H_1 \oplus H_2) \parr G_2$ and ($G_1 \parr H_1 \parr G_2 \ll I$ or $G_1 \parr H_2 \parr G_2 \ll I$);
- $G = G_1 \parr \exists x H \parr G_2$ and $G_1 \parr H[z/x] \parr G_2 \ll I$ and $z \notin var(G_1) \cup var(H) \cup var(G_2)$.

∎

We associate to the $\mathcal{LC}$ program P the operator T_P defined on $\mathcal{P}(\mathcal{M}_V)$

$$T_P(I) = \{(A, B)\theta | G \multimap A \in P, mgu(G, G') = \theta, B \parr G' \ll I\}$$

As we can see the relation $\ll$ has been introduced to take into account the complex structure of the body of $\mathcal{LC}$ clauses. Following the example of S-semantics for Horn clauses, T_P defines a bottom-up inference based on unification. The following theorem allows us to define a fixpoint semantics for $\mathcal{LC}$ programs.

Theorem 6 *The T_P operator is continuous on $(\mathcal{P}(\mathcal{M}_V), \subseteq)$. Then there exists the least fixed point $T_P \uparrow \omega$ of T_P.*

Definition 10
The fixpoint semantics of a $\mathcal{LC}$ program P is defined as $\mathcal{F}(P) = T_P \uparrow \omega$. ∎

Theorem 7 $O(P) = \mathcal{F}(P)$.

This semantics is certainly more adequate to model concrete features of $\mathcal{LC}$ computations such as computed substitutions. We think this can be certainly useful in view of static analysis of $\mathcal{LC}$ programs. For example we are allowed to inherit a good part of the methods developed in [6] for Horn clause logic languages.

6 Related work and conclusions

As already mentioned several languages have been proposed which use linear logic as their underlying logic. Miller [18, 25] uses the concept of uniform proof to characterize computationally interesting fragments of linear logic. In [18] the fragment is included in intuitionistic linear logic. Its main goal is to refine the language of hereditary Harrop Formulae, by exploiting the ability of linear logic to treat limited resource. The differences with our framework lies essentially in the absence of a mechanism of multiple head clauses, whence our fragment lacks a mechanism for the dynamic loading of modules. The areas of application seem indeed quite different. The results in [25] are more closely related to ours. In fact the fragment used to express the π-calculus as a theory of linear logic is essentially a higher order version of $\mathcal{LC}$. However the emphasis is on the use of the fragment as a metatheory of the π-calculus rather than as a logic programming language.

Other related results are [3, 4] and [2]. [3, 4] present Linear Objects (LO), an object-oriented logic programming language based on the proof theory of linear logic. LO can express the concurrent evolution of multiple objects, having complex states (essentially multisets of *slots*). Multiple-head clauses are exploited to express the evolution of our multisets of agents. Our language can be thought of as specifying the evolution of a single object without the powerful knowledge structuring of LO programs. However, in $\mathcal{LC}$ more sophisticated operations on the "single object" can easily be made available. In [2], LinLog a fragment of linear logic is presented. This language allows a compact representation of the so called *focusing* proofs. It shows

that LinLog does not lose expressive power w.r.t. linear logic. Every linear formula can be translated to LinLog preserving its focusing proofs. The $\mathcal{LC}$ language is essentially a fragment of LinLog. It is a compact representation of an *asynchronous* fragment (the $(\parr, \bot, \top)$ fragment) with some synchronous additions (the connectives $\oplus$ and 1). We believe that $\mathcal{LC}$ characterizes a class of applications for a subset of LinLog.

Finally we want to mention the relation to [7] and [9]. We think that the $\mathcal{LC}$ framework can easily be related to the general model of multiset rewriting. Indeed $\mathcal{LC}$ computations realize in a very natural way the *chemical metaphor* (as first noted in [17] for a smaller fragment). The multiset are *solutions* in which the *molecules* (agent or messages) can freely move. The *heating* of the solution (the application of the derivation rules) makes the molecules to be *decomposed* until they become simple *atoms*. At this point they are *ions* which by *chemical reactions* (the applications of the rules of the program) form new molecules.

Traditional concurrent logic programming languages ([11, 29, 30]) are quite distant relatives of our language. However we think that $\mathcal{LC}$ allows a more declarative view of concurrent interactions. For example we have not to constraint the unification to obtain synchronizations between parallel agents. Moreover the phase semantics does not seem to have the complexity of other declarative models of concurrent logic languages (see [12], for example).

In this paper we have presented a language for the concurrent programming based on the proof theory of linear logic. The $\mathcal{LC}$ framework can be characterized as an abstract logic programming language, which is closely related to the multiset rewriting computational paradigms like Chemical Abstract Machine and the Gamma model. We have proposed a semantics obtained as an instance of phase semantics of linear logic. Such a semantics describes the successful computations of $\mathcal{LC}$ programs. Finally we have given a fixpoint construction of a declarative semantics modeling answer substitutions. The $\mathcal{LC}$ language deserves further studies. We are currently investigating the possibility of extending the fragment of linear logic, so as to support mechanisms such as guards or hiding operators as proposed in [20].

An important thing to realize is that a program in the $\mathcal{LC}$ fragment is indeed a linear logical theory, while its computations are derivations in a linear proof system. Thus we need not to associate to the language $\mathcal{LC}$ a program logic. $\mathcal{LC}$ is its own program logic. We can use $\mathcal{LC}$ formulas to specify and investigate the properties of $\mathcal{LC}$ programs. We think however that the subject of specification and verification of $\mathcal{LC}$ programs still needs some work. Furthermore it would be interesting to establish to what extent the linear logic and its semantics can describe and model divergent or deadlocked computations (the present semantics simply ignores them). Moreover we think that applying some of the techniques suggested in [7] to build Γ programs we can define a set of tools for the synthesis of $\mathcal{LC}$ programs.

Finally we think an intriguing subject could be the study of abstract interpretations for $\mathcal{LC}$ programs. As shown in [20] we can start by taking as abstract domain a non complete phase model. We think that the work made in [19] regarding an abstract interpretation of Linear Objects could be quite helpful. Moreover the framework showed in [8] could be certainly useful.

Acknowledgements

The author would like to thank Giorgio Levi for his encouragement and support. Thanks are due to Alessio Guglielmi, for valuable comments, and to Andrea Masini, for helpful discussions.

References

[1] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1 & 2):3–59, 1993.

[2] J. M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[3] J. M. Andreoli and R. Pareschi. Linear Objects: logical processes with built-in inheritance. In D. H. D. Warren and P. Szeredi, editors, *Proc. Seventh Int'l Conf. on Logic Programming*, pages 495–590. The MIT Press, Cambridge, Mass., 1990.

[4] J. M. Andreoli and R. Pareschi. Communication as fair distribution of knowledge. In *Proc. of OOPSLA'91*, pages 212–229, 1991.

[5] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[6] R. Bagnara, R. Giacobazzi, and G. Levi. An Application of Constraint Propagation to Data-flow Analysis. In *Proc of Ninth IEEE Conference on AI Applications*, pages 270–276. IEEE Computer Society Press, 1993.

[7] J-P. Banâtre and D. Le Metayer. The gamma model and its discipline of programming. *Science of Computer Programming*, 15(1):55–77, 1990.

[8] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.

[9] G. Berry and G. Boudol. The chemical abstract machine. In *Proc. Seventeenth Annual ACM Symp. on Principles of Programming Languages*, pages 81–94, 1990.

[10] A. Brogi. And-parallelism without shared variables. In D. H. D. Warren and P. Szeredi, editors, *Proc. Seventh Int'l Conf. on Logic Programming*, pages 306–319. The MIT Press, Cambridge, Mass., 1990.

[11] K. L. Clark and S. Gregory. PARLOG: parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8:1–49, 1986.

[12] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. Semantic models for concurrent logic languages. *Theoretical Computer Science*, 86:3–33, 1991.

[13] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[14] M. Falaschi, G. Levi, and C. Palamidessi. A Synchronization Logic: Axiomatics and Formal Semantics of Generalized Horn Clauses. *Information and Control*, 60(1):36–69, 1984.

[15] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–113, 1985.

[16] J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[17] A. Guglielmi and G.Levi. Chemical logic programming? In D. Saccà, editor, *Proc. Eight Italian Conference on Logic Programming*, pages 39–51, 1993.

[18] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 32–42, 1991.

[19] R. Pareschi J. M. Andreoli, T. Castagnetti. Abstract Interpretation of Linear Logic Programming. In D. Miller, editor, *Proc. 1993 Int'l Symposium on Logic Programming*, pages 295–314. The MIT Press, Cambridge, Mass., 1993.

[20] N. Kobayashi and A. Yonezawa. Asynchronous communication model based on linear logic. Technical report, Department of Information Science, University of Tokyo, July 1992.

[21] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[22] Y. Lafont. Interaction nets. In *Proc. Fifth IEEE Symp. on Logic In Computer Science*, pages 95–108, 1990.

[23] G. Levi and F. Sirovich. A problem reduction model for non-independent subproblems. In *Proc. Fourth International Joint Conference on Artificial Intelligence*, pages 340–344, 1975.

[24] G. Levi and F. Sirovich. Generalized and-or graphs. *Artificial Intelligence*, 7:243–259, 1976.

[25] D. Miller. The π-calculus as a theory in linear logic: Preliminary result. In E. Lamma and P. Mello, editors, *Proc. of Third International Workshop on Extensions of Logic Programming*, volume 660 of *Lecture Notes in Computer Science*, pages 242–264. Springer-Verlag, Berlin, 1992.

[26] D. Miller, F. Pfenning, G. Nadathur, and A. Scedrov. Uniform proofs as a foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[27] L. Monteiro. An extension to horn clause logic allowing the definition of concurrent processes. In G. Goos and J. Hartmanis, editors, *Proc. of International Colloquium on Formalization of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

[28] L. Monteiro. A proposal for distributed programming in logic. In J. A. Campbell, editor, *Implementations of Prolog*, pages 329–340. Ellis-Horwood, 1984.

[29] E. Y. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[30] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 140–156. The MIT Press, Cambridge, Mass., 1987.

El λ-cálculo Etiquetado Paralelo (LCEP)*

Salvador Lucas† Javier Oliver†

Resumen

Presentamos un nuevo cálculo para la modelización de sistemas paralelos, el λ-cálculo Etiquetado Paralelo [18]. El formalismo surge de una propuesta inicial de H. Aït-Kaci [1,2], el Label-Selective λ-calculus, que describe un lenguaje, extensión del λ-cálculo [3], en el que los argumentos de las funciones se seleccionan mediante etiquetas. El conjunto de etiquetas incluye tanto posiciones numéricas como símbolos. En el Label-Selective λ-cálculo se refleja un paralelismo implícito entre la ejecución de los diferentes canales pero no existe la posibilidad de elección bajo un mismo canal. En este trabajo extendemos su sintaxis introduciendo el no determinismo en la evolución del sistema para reflejar así el comportamiento de los sistemas paralelos. La inclusión de nuevos operadores (el de paralelismo $\|$, el de secuencialidad $\circ$, el de elección no determinista $+$ y el de replicación !) y la introducción de nuevos conceptos (en particular, el concepto de túnel) permite a nuestro cálculo expresar con facilidad el paralelismo.

Palabras clave: Paralelismo, Extensiones del λ-cálculo, Algebras de Procesos, Programación Funcional.

1 Introducción

A principios de los años treinta, Church construyó el λ-cálculo libre de tipos [7]. Los fundadores del λ-cálculo [7] y la teoría de la lógica combinatoria [8,9] (relacionada con este cálculo) tenían dos ideas en mente: desarrollar una teoría general de las funciones computables y extender esta teoría para hacerla servir como un soporte uniforme para la lógica y una parte de las matemáticas. Sin embargo, el descubrimiento de distintas paradojas (Kleene y Rosser [14] demostraron que el sistema original de Church era inconsistente) hizo que no tuviera éxito. A pesar de ello, una parte importante de la teoría ha resultado relevante como base para la teoría de la computación. Gracias al análisis realizado por Turing [21], se puede afirmar que, a pesar de que su sintaxis es muy simple, el λ-cálculo es lo suficientemente potente para describir todas las funciones computables. La computación que captura el λ-cálculo, como mostró

*Este trabajo ha sido parcialmente subvencionado por CICYT, TIC 92-0793-C02-02

†Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Apdo. 22012, 46.020 Valencia, Spain, *e-mail: slucas(joliver)@dsic.upv.es*