Appears in *New Generation Computing, 15(4), 1997*

Static Analysis of Linear Logic Programming

Jean-Marc Andreoli Tiziana Castagnetti
Remo Pareschi
Rank Xerox Research Centre, Grenoble, France

Abstract

Linear Logic is gaining momentum in computer science because it offers a unified framework and a common vocabulary for studying and analyzing different aspects of programming and computation. We focus here on models where computation is identified with proof search in the sequent system of Linear Logic. A proof normalization procedure, called "focusing", has been proposed to make the problem of proof search tractable. Correspondingly, there is a normalization procedure mapping formulae of Linear Logic into a syntactic fragment of that logic, called LinLog, where the focusing normalization for proofs can be most conveniently expressed. In this paper, we propose to push this compilation/normalization process further, by applying abstract interpretation and partial evaluation techniques to (focused) proofs in LinLog. These techniques provide information concerning the evolution of the computational resources (formulae) during the execution (proof construction). The practical outcome that we expect from this theoretical effort is the definition of a general tool for statically analyzing and reasoning about the runtime behavior of programs in frameworks where computations can be accounted for in terms of proof search in Linear Logic.
Keywords: Linear Logic, sequent systems, proof search, static analysis, abstract interpretation, partial evaluation.

1 Introduction

Linear Logic [29] is gaining momentum in computer science because it offers a unified framework and a common vocabulary for studying and analyzing different aspects of programming and computation. Indeed, by providing accounts for the notions of *computation-as-proof-normalization* and of *formulae-as-types*, Proof Nets (the Linear Logic machinery for natural deduction) allow capturing the theoretical bases of functional programming [1] in a way which, with respect to more traditional approaches, is at the same time more faithful to the practice of computing (for Intuitionistic Linear Logic gives direct insight on control issues in sequential functional computations) and more innovative (for Classical Linear Logic lays the foundation for parallel functional programming). But a similar impact has also been achieved on the side of the *computation-as-proof-search* paradigm, which gives the background for this paper. Here we find developments such as: declarative reconstructions of concurrent computational models (e.g. process calculus with ACL [35, 36], concurrent object oriented programming with LO [5], concurrent constraints [50]); planning [42]; natural language parsing [21, 22]; logical accounts of the controversial negation-as-failure inference rule of logic programming [14]; logical account of exception mechanism in terminological hierarchies [25]; type and mode inference in logic programs [49].

This flowering of developments has been backed by theoretical contributions concerning those aspects of Linear Logic which are essential to proof search. In particular, there have been several investigations of the crucial problem of permutations of inference figures during proof construction [2, 6, 31, 45, 26]. Other contributions have investigated in terms of proof search the complexity of different fragments of Linear Logic [38]. In this paper, we contribute to the study of yet another aspect: the abstract interpretation of Linear Logic proofs. The practical outcome that we expect from this theoretical effort is the definition of a general tool for statically analyzing and reasoning about the runtime behavior of programs in frameworks where computations can be accounted for in terms of proof search in Linear Logic. Various activities related to the support, maintenance and optimization of programs call for such analysis facilities: program verification, program transformation (based on behavioral equivalence), debugging, visualization of execution trace, compilation optimization, etc. We focus here mainly on the last aspect, namely the compile-time optimization of the search process.

The rest of the paper is structured as follows. In Section 2, we recall the basic proof system on which we rely throughout the paper. Section 3 introduces a static analysis technique capable of inferring basic properties

of proofs in this system. Section 4 shows how to infer information not only on the proofs themselves but also the proof search process; this of course requires minimal assumptions on the proof search mechanism. Section 5 shows how these general results can be exploited for various compile-time optimizations in different programming frameworks which can be modelled as proof search in our system. Section 6 discusses related works and Section 7 hints at future developments.

2 Proof Search in Linear Logic

2.1 Proof Search in Sequent Systems

We identify here computations with proof search. More precisely, the universe of computation is given by an *incomplete* proof Π (i.e. a proof tree which contains open leaves), and the computation (i.e. the proof search activity) basically consists of expanding this proof (concurrently) at each of its open leaves by application of the inference figures of the sequent system (here, the sequent system of Linear Logic).

A proof search procedure is intrinsically non-deterministic, in that the criterion for the selection of the instance of inference figure to apply at each step is not specified. Of course, all the different possible choices could be enumerated, and explored, sequentially or in parallel. However, it appears that many such alternative choices are irrelevant and are only artificial by-products of the intrinsic and unnecessary sequential nature of sequent proofs (and of the search procedure). Indeed, many (but not all) inference figures of the sequent system are permutable, so that the different choices in the order of application of such inference figures only lead to irrelevant syntactic differences in the proof.

In [2], some proof search strategies are introduced, which eliminate many (but not all) of the irrelevant choices. These strategies are described by refining the sequent system of Linear Logic, using sequents which have a more sophisticated structure than simple multisets. The idea is to include in the representation of the sequents some information about the status of the search. This information is then used to preclude some redundant or useless choices which would otherwise be allowed in the standard sequent system. This technique, also called "focusing", is shown (in [2]) to be sound and complete, in the sense that any provable formula of Linear Logic has a "focusing" proof (more precisely, any correct proof of Linear Logic can be mapped into a focusing proof by simple permutations of the inference figures).

2.2 LinLog: a Normal Form for Linear Logic

Although the focusing strategy has been devised for proof search directly in the sequent system of Linear Logic, it is more conveniently expressed within a syntactic fragment of that logic called LinLog [2], to which any formula of Linear Logic can be mapped. Thus, LinLog defines a kind of normal form for formulae in Linear Logic. Its status is quite similar to that of clauses in Classical Logic: LinLog and focusing proofs can be viewed as the true generalization to the framework of Linear Logic of clauses and resolution proofs in Classical Logic.

LinLog syntax consists of two classes of formulae, *Goals* and *Methods*, defined from the class A of positive atomic formulae (simply called atoms):

- Goals are positive formulae (containing no dual of atom) such that no synchronous connective occurs in the scope of an asynchronous one[1]. We use here the terminology of [2] where the connectives $\bot, \parr, ?, \top, \&, \forall$ are called "asynchronous" and the (dual) connectives $1, \otimes, !, 0, \oplus, \exists$ are called "synchronous". A goal is dubbed asynchronous or synchronous depending on its topmost connective. Notice that, consequently, asynchronous goals cannot contain synchronous connectives.

- Methods are closed formulae of the form

$$\forall \vec{x}\ (G \multimap A_1 \parr \cdots \parr A_r)$$

  where G is a goal (called the body of the method) and $A_1, \ldots, A_r$ is a multiset of atoms ($r \geq 1$), called the head of the method. The symbol $\multimap$ is the Linear implication, defined by $F \multimap G \stackrel{\text{def}}{=} F^{\perp} \parr G$. Methods are often written in a simplified notation as:

$$A_1 \parr \cdots \parr A_r \circ\!\!- G$$

Definition 1 *A program is a set of methods and a context is a multiset of goals with at most one synchronous goal.* LinLog *sequents are of the form $\mathcal{P} \vdash \mathcal{C}$ where $\mathcal{P}$ is a program and $\mathcal{C}$ is a context.*

[1] In fact, the definition is a bit more complex when exponentials are involved; see [2] for details.

In the LinLog fragment, the focusing strategy can be (informally) described as follows.

- If the context of the current sequent contains a synchronous goal, this goal (which is unique by definition of LinLog contexts) must be immediately decomposed, using an instance of the inference figure corresponding to the top-most connective of that synchronous goal.
- If the context of the current sequent contains only asynchronous goals, they must be recursively decomposed, in any order (possibly in parallel): order is irrelevant in this case.
- If the context of the current sequent is flat, i.e. consists only of atoms, then a (ground) instance of a method of the program must be selected, such that its head matches a sub-multiset of the context. This sub-multiset is then *replaced* by the body of the selected method [2].

The following theorem (proved in [2]) ensures soundness and completeness of the LinLog strategy.

Theorem 1 *Let $\mathcal{P}$ be a program and G be a goal. The sequent $\mathcal{P} \vdash G$ is derivable in* LinLog *if and only if the sequent $\vdash (!\mathcal{P})^{\perp}, G$ is derivable in Linear Logic.*

An algorithm mapping any formula of Linear Logic into LinLog is described in [2]. It provides a transformation to normal form for Linear Logic formulae, analogous to the clausal form in Classical Logic.

Theorem 2 *Any formula F of Linear Logic can be mapped into a pair $\langle \mathcal{P}; G \rangle$, where $\mathcal{P}$ is a* LinLog *program and G a goal, such that $\vdash F$ is derivable in Linear Logic if and only if $\mathcal{P} \vdash G$ is derivable in* LinLog.

This normalization mapping is of linear complexity in its input, just like normalization to clausal form in Classical Logic. However, we have a more specific result in Linear Logic, which has no counterpart in Classical Logic: there is an *isomorphism* between the *focused* proofs of a formula in Linear Logic and the LinLog proofs of its normal form. In other words, the problem of proof search in Linear Logic, when restricted to focused proofs, is isomorphic to the problem of proof search in LinLog and can be reduced to it without loss of generality. Thus, LinLog provides the framework for Logic Programming systems (where the computational paradigm is proof search) based on focused proofs in Linear Logic. All the systems mentioned in the introduction belong to that category.

2.3 **Simple** LinLog

In fact, for simplicity sake, in the rest of the paper, we focus on a subfragment of LinLog, which has proved useful for modeling many computational phenomena, and which we call here "Simple" LinLog. The techniques presented here could be straightforwardly extended to full LinLog (and, hence, to full Linear Logic by theorem 2).

In Simple LinLog, the body of methods are Simple goals, given by the following syntax:

$$G \;=\; A \mid G \parr G \mid \top \mid G \,\&\, G$$

The LinLog strategy for this subfragment can be captured by the following specialized inference system ($\mathcal{P}$ is a program and $\mathcal{C}$ is a context; G, G_1, G_2 are Simple goals):

- Decomposition inference figures (applied deterministically in any order to non flat contexts):

$$[\parr]\ \frac{\mathcal{P} \vdash \mathcal{C}, G_1, G_2}{\mathcal{P} \vdash \mathcal{C}, G_1 \parr G_2} \qquad [\top]\ \frac{}{\mathcal{P} \vdash \mathcal{C}, \top} \qquad [\&]\ \frac{\mathcal{P} \vdash \mathcal{C}, G_1 \quad \mathcal{P} \vdash \mathcal{C}, G_2}{\mathcal{P} \vdash \mathcal{C}, G_1 \,\&\, G_2}$$

- Progression inference figure (applied non-deterministically to flat contexts):

$$[\circ\!\perp]\ \frac{\mathcal{P} \vdash \mathcal{C}, G}{\mathcal{P} \vdash \mathcal{C}, A_1, \ldots, A_r}$$

  where $A_1 \parr \cdots \parr A_r \circ\!\perp G$ is a ground instance of a method of the program $\mathcal{P}$.

Given that Decomposition can be applied deterministically, the inference system above can be compiled into an equivalent simplified system using a *unique* inference figure. We first define by induction the set $\|G\|$ of "par-components" of a goal G.

$$\begin{array}{lcl} \|G\| & = & \{\{A\}\} \text{ if } G \text{ is an atom } A \\ \|G_1 \,\&\, G_2\| & = & \|G_1\| \cup \|G_2\| \\ \|G_1 \parr G_2\| & = & \{C_1 \uplus C_2 \text{ s.t. } C_1 \in \|G_1\| \text{ and } C_2 \in \|G_2\|\} \end{array}$$

Each par-component of G is a multiset of atoms, corresponding to one branch of proof in the decomposition of G. The inference system of LinLog can then be based on the following single inference figure (called the LinLog inference figure):

[2] Things are slightly more complex when the exponentials are involved; see [2] for details.

$$[\text{LinLog}]\ \frac{\mathcal{P} \vdash \mathcal{C}, C_1 \quad \cdots \quad \mathcal{P} \vdash \mathcal{C}, C_n}{\mathcal{P} \vdash \mathcal{C}, A_1, \ldots, A_r}$$

where $A_1 \parr \cdots \parr A_r \circ\!\!- G$ is a ground instance of a method of the program $\mathcal{P}$, and $\|G\| = \{C_1, \ldots, C_n\}$.

Important remarks:

- In the sequel, the word "proof" will always refer to a proof in the sequent system based on the single inference figure above (except stated otherwise).
- The program $\mathcal{P}$ in a proof, (and *a fortiori* its set of ground instances), need not be finite. We only assume that there is a finite bound on the complexity of the methods of $\mathcal{P}$, the complexity of a formula being the number of connectives it contains.

3 Static Analysis of LinLog Programs

3.1 Methodology

The transformation to LinLog normal form, provided by theorem 2, can be viewed as a normalization/compilation transformation for the problem of proof search in Linear Logic. But this kind of transformations can be pushed much further, by application of static analysis techniques to LinLog programs. Our intention here is to analyze LinLog proofs in order to infer information which can be used to optimize proof search in LinLog. However, in a first stage, we make strictly no assumption on the actual operational mechanism which implements the search. Consequently, although our work is inspired by the techniques which have already been developed for static analysis of logic languages based on proof search, such as Prolog, we have discarded from these techniques all the aspects which are dependent on the operational semantics, namely (*i*) the so-called "computation rule" (which selects a sub-node to expand at each step of the resolution) and (*ii*) the propagation of the variable instantiations by the mechanism of unification. It is interesting to notice that, while in Prolog, it is impossible to ignore these aspects, because the results of the analysis would then be too poor, in LinLog, one can still infer useful information by focusing exclusively on the proofs themselves and not the way they are searched. Section 4 introduces some basic assumptions on the operational mechanism of the search in order to infer more information.

The main idea we keep from the work on traditional proof search analysis in logic is that of a possibly infinite set of AND-trees (the possible proofs) being encoded in an AND-OR tree and then approximated into a finite graph. Finiteness is ensured by applying an appropriate *abstraction function*. The graph itself is obtained by *partially evaluating* the abstract program in all possible ways. Hence, our analysis involves the two traditional mechanisms of abstract interpretation and partial evaluation.

3.2 Abstract Interpretation

To be useful to our purposes, the design of the abstraction function must obey the following two requirements:

(*i*) the abstraction of a proof based on a concrete program must be a proof based on the corresponding abstract program;

(*ii*) the whole set of possible proofs based on the abstract program must be representable in a *finite* structure (called here a "partial evaluation graph").

3.2.1 The Abstraction Mapping

We give here a formal definition of the notion of abstraction mapping.

Definition 2 *An abstraction mapping is a function τ defined on the syntactic constructs of* LinLog *(atoms, goals, methods, programs, contexts, proofs etc.), such that*

- *The set $\mathcal{A}$ of abstract atoms, obtained by applying τ to concrete atoms, is finite.*
- *The abstraction $\tau(E)$ of a syntactic construct E is obtained by consistently replacing in E each occurrence of an atom A by its abstraction $\tau(A)$.*

An immediate consequence of this definition is that an abstraction mapping preserves the syntactic category of the constructs to which it applies. This is obvious for goals, methods, programs and contexts; it also holds for proofs since the notion of "proof" *commutes* with the notion of "abstraction". Indeed, the abstraction of a concrete proof based on a program $\mathcal{P}$ is an abstract proof based on the abstraction of the program $\mathcal{P}$. More formally:

Proposition 3 *Let* $\mathsf{Pr}(\mathcal{P})$ *be the set of proofs based on a program* $\mathcal{P}$*. Then, for all abstraction mappings* τ*,*

$$\tau(\mathsf{Pr}(\mathcal{P})) \subset \mathsf{Pr}(\tau(\mathcal{P}))$$

This property derives from the fact that each instance of the LinLog inference figure is mapped, by abstraction, into an instance of the same figure (using the abstraction of the applied method).

Another consequence of definition 2 is that the number of abstract formulae of a given complexity is finite. Hence, keeping into account that we have assumed that there is a global finite bound on the complexity of the methods in the concrete program, we can conclude that there is only a finite number of (abstract) methods in an abstract program.

Proposition 4 *The abstraction of a program is always a* finite *set (of abstract methods).*

But this does not mean that the number of proofs based on the abstract program is finite. In fact, there might be infinitely many possible abstract proofs based on the same finite abstract program, which we want to represent in a finite way, by *approximation*. The approximation mechanism, described in the next section, relies on the structure of the set of flat abstract contexts. Each flat abstract context is a finite multiset of abstract atoms, and hence, multiset inclusion induces a lattice structure on flat abstract contexts. In order to have a *complete* lattice, with a notion of limit and hence a mechanism for approximation, we work on "sorts" instead of finite multisets. A sort is taken here to be a possibly infinite multiset of abstract atoms.

Formally, let $\mathcal{A}$ be the (finite) set of abstract atoms. Let $\bar{\mathbf{N}} = \mathbf{N} \cup \{\infty\}$ be the set of natural integers together with the value "infinity".

Definition 3 *A sort is a mapping* $\psi : \mathcal{A} \mapsto \bar{\mathbf{N}}$*.*

For each abstract atom $a \in \mathcal{A}$, the number $\psi(a)$ denotes the (possibly infinite) number of occurrences of a in the sort ψ.

3.2.2 Properties of Sorts

Let Ψ_o be the set of sorts. It is equipped with an operation of addition and a partial order relation derived in a natural way from those of $\bar{\mathbf{N}}$: for all $\psi_1, \psi_2 \in \Psi_o$

$$\begin{array}{l} \forall a \in \mathcal{A} \;\; (\psi_1 + \psi_2)(a) \;=\; \psi_1(a) + \psi_2(a) \\ \psi_1 \preceq \psi_2 \;\text{ if and only if }\; \forall a \in \mathcal{A}, \psi_1(a) \leq \psi_2(a) \end{array}$$

The addition (which is identical to multiset union in the case of finite sorts) defines a monoid over Ψ_o and the partial order defines a complete lattice. Indeed,

Proposition 5 *Any set of sorts* $\Psi \subset \Psi_o$ *has a lowest upper bound* ($\sqcup\Psi$) *and a greatest lower bound* ($\sqcap\Psi$) *defined by*

$$\forall a \in \mathcal{A} \left\{ \begin{array}{l} (\sqcup\Psi)(a) = \sup\{\psi(a) \;\text{ s.t. }\; \psi \in \Psi\} \\ (\sqcap\Psi)(a) = \inf\{\psi(a) \;\text{ s.t. }\; \psi \in \Psi\} \end{array} \right.$$

The sup and inf operations are totally defined here, since any subset of $\bar{\mathbf{N}}$ has a greatest lower bound and a least upper bound.

Furthermore, we make use later on, in an essential way, of a well known property of the lattice of sorts:

Theorem 6 *For any infinite sequence* (ψ_n) *of sorts, there exists indices* p, q *such that*

$$p < q \;\text{ and }\; \psi_p \preceq \psi_q$$

This property is proved in appendix A. Notice the importance in the proof of the assumption that the set $\mathcal{A}$ of abstract atoms is finite. Without such an assumption, the theorem does not hold: if, for instance, $\mathcal{A}$ were $\mathbf{N}$, then a counterexample would be given by the sequence (ψ_n) where $\psi_n(n) = 1$ and $\psi_n(m) = 0$ for $m \neq n$.

3.2.3 The Canonical Abstraction Mapping

Traditionally, the atoms in concrete LinLog programs are "tagged" tuples of the form $p(a_1, \ldots, a_n)$ where the arguments $a_1, \ldots, a_n$ belong to a given set of values and the tag p belongs to a given set of predicates. Each predicate is supposed to have a fixed arity. This structure allows programs to be written as finite sets of program "moulds", which have the same syntax as methods except that atoms occur in the form $p(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ belong to a given set of variable names. A mould, built from such atoms, encodes the set of methods obtained by consistently replacing each variable name in the mould by a value (this operation is called instantiation).

Therefore, if the set of values is infinite, the concrete program, which consists of all the possible instantiations of the moulds from which it is defined, is also infinite. On the other hand, the complexity of the methods in the program is finitely bound by the maximal complexity of the (finite set of) moulds.

Another way of looking at moulds is to consider them as universally quantified formulae in first-order logic. Notice however that we do not allow function symbols nor individuals in programs. Indeed, it can easily be shown that methods using function symbols can be "flattened" (each occurrence of function symbols is replaced by a variable together with a new "built-in" predicate, added to the head of the method, linking this variable to the arguments of the function symbol). Furthermore, we want our static analysis to focus only on the distribution of resources in proofs with as little assumptions as possible on the structure of the resources, be they first-order terms or any other kind of data-structure. Let's stress again that in LinLog, as opposed for instance to Prolog, a lot can be achieved without this structural knowledge.

When the program is specified from a set of moulds, a natural choice for the abstraction function τ consists of mapping each atom into its predicate (with the associated arity); we shall refer to such a function as the *canonical* abstraction mapping[3]. Given that the set of moulds defining the program is finite and that instantiating a mould with values does not modify the predicates which occur in it, the possibly infinite set of atoms occurring in the concrete program is always mapped, by a canonical abstraction mapping, into a finite set of predicates. Therefore, the canonical abstraction mapping satisfies the criterion of definition 2.

3.3 Partial Evaluation

Having specified, in the previous section, the abstract interpretation mapping τ, we now come to the actual analysis of the proofs at the abstract level. Our aim is to represent in a finite way all the possible proofs of a given abstract program call. In other words, given a finite abstract program $\mathcal{P}$ and an initial abstract context, i.e. a sort ψ_o, we want to build a finite representation of the set of possible proofs of ψ_o with respect to $\mathcal{P}$.

3.3.1 The Total Evaluation Tree

There may be more than one proof with ψ_o as conclusion, corresponding to the selection of different methods in the LinLog inference figure. We encode all these choices (and hence, all the possible proofs) into a single structure, called the "total evaluation tree", which is a (usually infinite) *and-or* tree. Informally, the outgoing edges from or-nodes correspond to choices of applicable methods and those from and-nodes correspond to the decomposition of the body of the selected method. Formally, the total evaluation tree $\mathcal{T}_{\langle \mathcal{P};\psi_o \rangle}$ with respect to an abstract program $\mathcal{P}$ and a sort ψ_o is the smallest (usually infinite) tree verifying the following requirements:

- Each or-node is labeled with a sort. Each and-node is labeled with a pair $\langle \psi; G \rangle$ consisting of a sort ψ and a goal G.
- The root is an or-node labeled with ψ_o.
- From each or-node ψ and for each method with head ψ_h (strictly speaking, ψ_h is the multiset of atoms occurring in the head of the formula) and body G such that $\psi = \psi_h + \psi'$, there is an edge to an and-node $\langle \psi'; G \rangle$.
- From each and-node $\langle \psi; G \rangle$ and for each "par-component" ψ_r in $\|G\|$, there is an edge to an or-node $\psi + \psi_r$.

Total evaluation trees are characterized by the following two important properties:

Proposition 7 *For all abstract program P and sort ψ, the total evaluation tree $\mathcal{T}_{\langle \mathcal{P};\psi \rangle}$ has a bounded branching factor.*

This is a direct consequence of the fact that abstract programs contain only finitely many methods, and the body of each method has finitely many par-components. Therefore, both and-nodes and or-nodes in the tree have a bounded number of outgoing edges. As a corollary, we have

Corollary 8 *A total evaluation tree is infinite if and only if it has an infinite branch.*

This directly follows from the previous proposition by König's lemma, stating that any tree with a bounded branching factor is infinite only if it has an infinite branch.

The aim of the partial evaluation is to provide a finite representation for the possibly infinite total evaluation trees. The basic idea is to show that the infinite branches of a total evaluation tree always contain some regularity patterns and can thus be finitely encoded using these patterns.

[3] We can of course think of situations where tuples are "untagged", i.e. are not associated with any predicate name; this is for instance the case in the Gamma model [8], or in Linda [13]. In this case, the canonical abstraction function would provide no useful information.

$p \circ\!\!- r$
$q \circ\!\!- r$
$r \circ\!\!- \top$
$r \circ\!\!- p \;\&\; q$

A Prolog-like program.

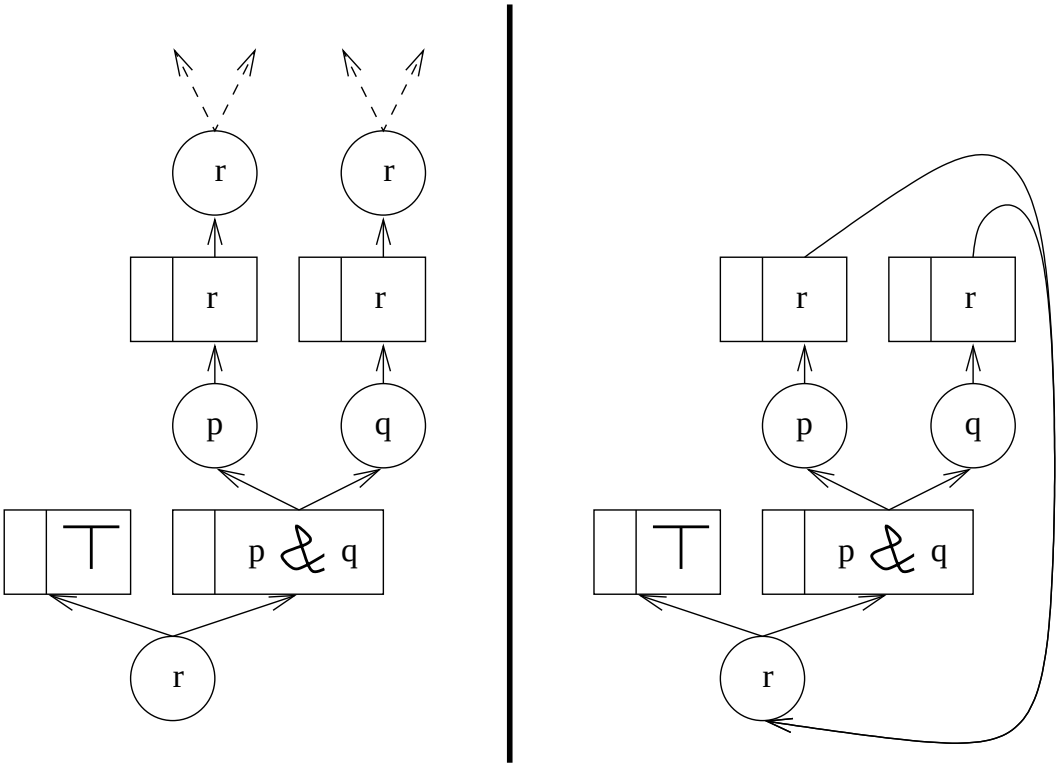

Figure 1: The infinite total-evaluation tree (left) is represented as a finite partial-evaluation graph (right)

An elementary illustration of this technique is given by the case of a program containing no occurrence of the connective $\parr$, and where the initial sort is reduced to a single atom (see Figure 1); thus, the situation reminds directly of a query in a Prolog-like language. In this case, it is easy to see that each or-node in the total evaluation tree is labeled with a *single* atom. As the number of atoms is finite, an infinite branch in this tree contains necessarily two or-nodes ν, ν' labeled with the same atom. Now, the subtree at ν is identical to the subtree at ν'. Hence, the incoming edge at ν' can be redirected to ν (assuming ν is closer to the root than ν'), and the infinite branch disappears. Thus, the possibly infinite total evaluation tree is mapped into a finite partial evaluation graph, where infinite branches are replaced by cycles (see Figure 1). Of course, in this Prolog-like case given here only for explanatory purpose, the resulting graph contains no useful information.

We make the following conventions for the graphical representation of the evaluation trees: an or-node labeled with a sort ψ is represented as a round box containing ψ; an and-node labeled with the pair $\langle \psi; G \rangle$ is represented as a square box divided in two zones by a vertical bar: ψ appears on the left-hand side while G appears on the right-hand side of the bar. In the example of Figure 1, the structure of the program is such that the left-hand side ψ at an and-node is always empty, but, obviously, this does not hold in general.

3.3.2 The Partial Evaluation Graph

When the connective $\parr$ occurs in the program, the situation is more complex than in the case of Figure 1, although it bears some similarity. Indeed, as the set of sorts is not finite, there might be infinite branches in which all the or-nodes are labeled with pairwise different sorts, and hence, we cannot simply "short-cut" such branches as in Figure 1. However, by considering the sequence of labels of or-nodes along infinite branches, Theorem 6 informs us that there exist necessarily two distinct or-nodes ν, ν' on each infinite branch such that (i) node ν is closer to the root than ν' and (ii) the sorts ψ, ψ' labeling, respectively, ν and ν', satisfy the property $\psi \preceq \psi'$ (hence $\psi' = \psi + \psi_e$ for some ψ_e).

This means that the sub-tree at ν' can be obtained from the sub-tree at ν by adding sort ψ_e to all the nodes, and by possibly grafting new branches on to some or-nodes, corresponding to methods newly enabled by the atoms added with ψ_e. Thus a new form of regularity appears, which we exploit to build a finite representation of the infinite branch.

Definition 4 *A cycle node in a total evaluation tree is an or-node ν', labeled with a sort ψ', such that there exists an or-node ν closer to the root on the same branch, and labeled with a sort ψ such that $\psi \preceq \psi'$. The node ν closest to the root and satisfying this property is then called the basis of the cycle node ν'. If the labels of ν and ν' are different (hence $\psi \prec \psi'$), then the cycle node ν' is said to be strict.*

We build the finite representation of the total evaluation tree in two steps:

1. We first build the "partial evaluation tree" which is identical to the total evaluation tree except that the subtrees at the cycle nodes are pruned. The algorithm for building the partial evaluation tree is given in Figure 3. It incrementally expands the nodes in the usual way, but checks, whenever a new node is installed, whether it is a cycle node: in this case, the node is not expanded. For example, in the total evaluation tree of Figure 2 (left), the or-node q, r, s is a (strict) cycle node, whose basis is the or-node q, r. Hence, in the partial evaluation tree, the subtree at node q, r, s (denoted on the figure with dashed lines) is not built. The same applies to the non-strict cycle node r, u.

2. The "partial evaluation graph" is then obtained from the partial evaluation tree by elimination of the cycle nodes (which have not been expanded). Non-strict cycle nodes (which, by definition, have the same label as their basis) are eliminated in the usual way by redirecting their incoming edge to their basis. In the example of Figure 2, that would be the case of the cycle node r, u. The treatment of strict cycle nodes is a bit more complex and relies on the idea of "generalization" explained below. A similar idea has been developed independently in the context of Petri Nets in [48]. Notice however that Petri Nets can only handle the connective $\parr$ of Simple LinLog, not the &, let alone the other connectives of full LinLog.

3.3.3 The Generalization Step

Let ν' be a strict cycle node (labeled with ψ') and let ν be its basis (labeled with ψ). By definition, there is a non empty sort ψ_e such that $\psi' = \psi + \psi_e$. The sequence of methods applied from ν to reach ν' can also be applied from ν', except that the label at each node in the sequence will now be augmented by ψ_e, thus leading to another or-node labeled with $\psi + 2\psi_e$. By reiterating this operation, we show that there is always an infinite branch stemming from any cycle node in the total evaluation tree, and containing a sequence (ν_n) of nodes such that each ν_n is labeled with $\psi + n\psi_e$.

The subtree at ν_{n+1} is derived from the subtree at ν_n by adding ψ_e to all the nodes and possibly grafting new branches enabled by these added atoms. Clearly, as the number of edges stemming from an or-node is bounded

$p \circ\!\!\perp q \parr r$
$q \circ\!\!\perp (s \parr t) \,\&\, u$
$r \parr s \circ\!\!\perp \top$
$r \parr t \circ\!\!\perp r \parr q$
$r \parr u \circ\!\!\perp r \parr u$

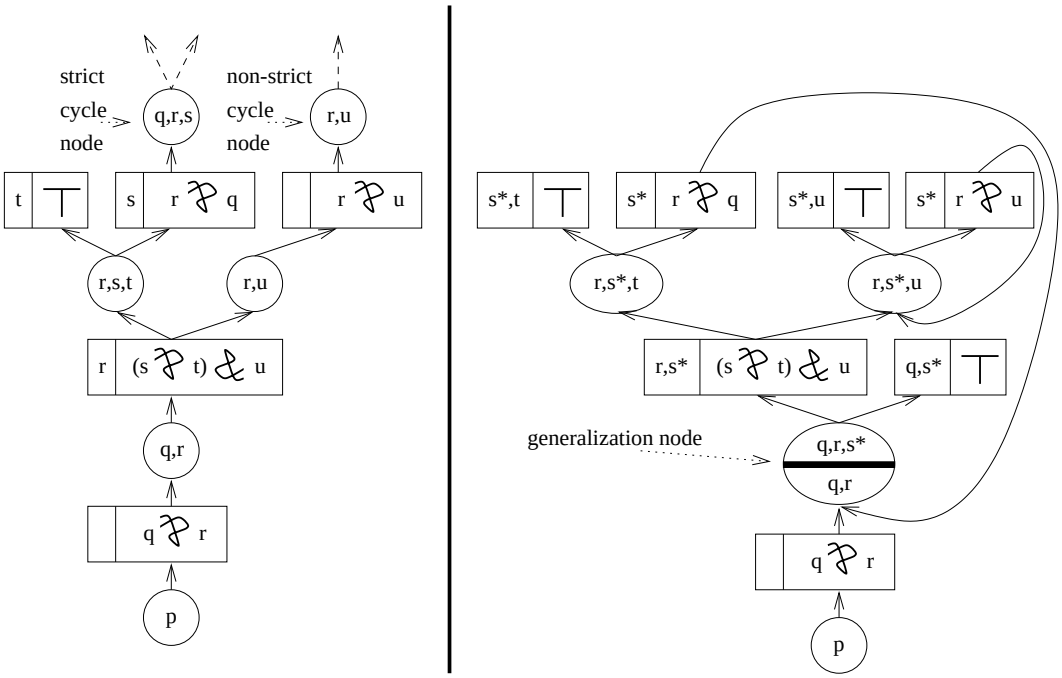

Figure 2: The infinite total-evaluation tree (left) is represented as a finite partial-evaluation graph (right)

Function PEVTREE: **returns** the partial evaluation tree
  Let Π be the tree reduced to a single or-node ν_o labeled with ψ_o;
  Call EXPAND(ν_o);
  Return Π;

Procedure EXPAND(ν: open or-node in Π)
  Let ψ be the sort labeling ν;
  If there exists an or-node closer to the root on the same branch as ν
      and labeled with a sort $\psi_b \preceq \psi$ **Then**
    Mark ν as a cycle node;
  Else Foreach method M in the program **Do**
    Let ψ_h be the head of M and G its body;
    If $\psi = \psi_h + \psi'$ for some sort ψ' **Then**
      Create an edge from ν to a new and-node ν_G labeled with $\langle \psi'; G \rangle$;
      Foreach par-component ψ_g of G **Do**
        Create an edge from ν_G to a new or-node ν' labeled with the sort $\psi' + \psi_g$;
        Call EXPAND(ν');

Figure 3: The algorithm for building the partial evaluation tree

Function PEVGRAPH: **returns** the partial evaluation graph
  $\Pi \leftarrow$ PEVTREE();
  While Π contains some *strict* cycle node **Do**
    Select one strict cycle node ν' in Π;
    Call GENERALIZE(ν');
  Redirect the incoming edge of each *non-strict* cycle node in Π to its basis;
  Return Π;

Procedure GENERALIZE(ν': strict cycle node in Π)
  Let ν be the basis of the cycle node ν';
  Let ψ, ψ' be the labels at, respectively, ν and ν';
  By hypothesis, $\psi \prec \psi'$, hence there exists a sort ψ_e such that $\psi' = \psi + \psi_e$;
  Discard the subtree at ν and change the label of ν to $\psi + \psi_e^*$;
  Call EXPAND(ν);

Figure 4: The algorithm for building the partial evaluation graph

by the (finite) number of methods in the program, beyond a sufficiently large value of n, no grafting is needed, so that the subtree at ν_{n+1} is strictly isomorphic to that at ν_n ("modulo" ψ_e). When $n \to \infty$, the adjunction of ψ_e becomes negligible, so that the subtree at ν_n is identical to the subtree at ν_{n+1} and this regularity can be captured in the usual way by redirecting the incoming edge of ν_{n+1} to ν_n. The infinite branch of the total evaluation tree can thus be finitely represented in the partial evaluation graph by an "asymptotic" cycle.

To formalize this idea, we introduce the following operation on sorts.

Definition 5 *For any sort ψ, we take ψ^* to be the sort containing infinitely many occurrences of each element of ψ. Formally:*

$$\forall a \in \mathcal{A} \qquad \psi^*(a) \;=\; \text{if } (\psi(a) = 0) \text{ then } 0 \text{ else } \infty$$

Notice that any sort ψ satisfies $\psi^* + \psi = \psi^*$. We can now formulate the treatment of strict cycle nodes in the partial evaluation tree: let ψ' be the label of a strict cycle node whose basis ν is labeled with ψ (hence $\psi' = \psi + \psi_e$ for some non empty ψ_e). The node ν initiates the infinite sequence of nodes (ν_n) which we collapse into a single node labeled with the lowest upper bound of all the labels of the nodes ν_n. Given that each ν_n is labeled with the sort $\psi + n\psi_e$, the generalization label is given by

$$\bigsqcup_n (\psi + n\psi_e) \;=\; \psi + \psi_e^*$$

The node ν is re-labeled with this generalized label and the subgraph at ν is recomputed. The sequence of methods leading from ψ to ψ' will now lead from $\psi + \psi_e^*$ to $\psi' + \psi_e^*$ but these two sorts are identical. Indeed,

$$\psi' + \psi_e^* = (\psi + \psi_e) + \psi_e^* = \psi + (\psi_e + \psi_e^*) = \psi + \psi_e^*$$

Hence the strict cycle node ψ' has been mapped in this way into a non-strict one which can be eliminated in the usual way, by introducing a cycle in the graph. The partial evaluation graph is obtained by recursively applying this generalization procedure to all the strict cycle nodes of the partial evaluation tree.

The formal specification of the algorithm for building the partial evaluation graph is given in Figure 4. Figure 2 (right) shows the result obtained with a sample program. The leaf q, r, s in the partial evaluation tree is a strict cycle node with basis q, r (here $\psi_e = s$). Hence the basis node q, r is generalized and re-labeled with q, r, s^* (the old and the new label are both shown on the figure). Now, the expansion of the newly labeled node is recomputed and yields only non-strict cycle nodes q, r, s^* and r, s^*, u which result in the two cycles of the graph (Figure 2 right).

Theorem 9 *The algorithms* PEVTREE *(Figure 3) and* PEVGRAPH *(Figure 4) always terminate.*

Demonstration: It is plain that the algorithm building the partial evaluation tree (Figure 3) terminates, since by theorem 6, it must reach a cycle node (or a termination node) on all branches after a finite number of expansions, and hence stop. The algorithm building the partial evaluation graph (Figure 4) also terminates, since its only possible source of non termination is the loop which performs the generalizations; but this loop must stop after a finite number of performances. Indeed, the number of possible generalizations along a branch is bounded by the

(finite) number of atoms: once generalization has been performed on all the atoms, we obtain a node labeled with the maximal sort $\mathcal{A}^*$ (where all atoms have infinitely many occurrences) and the expansion of this node yields only non-strict cycle nodes, since application of any method on $\mathcal{A}^*$ yields $\mathcal{A}^*$ itself. Of course, the partial evaluation graph obtained in this case is not very informative. □

Notice that the algorithm for building the partial evaluation graph (Figure 4) could easily be modified for improved efficiency. For example, in a generalization step, it would be more efficient to try to reuse (at least in part) the already computed expansion of the basis of the cycle node, instead of expanding from scratch the generalized node. Another improvement could come from a subsumption test: it is of no use to expand a node if a larger (or identical) node has already been expanded in another branch. Finally, the computation of the partial evaluation graph can be made parallel by exploring all the branches in parallel. This is always possible since the final resulting graph does not depend on the order of the generalization. Our procedure can thus be viewed as a form of parallel compilation.

3.4 A Note on Complexity

It is difficult to estimate the complexity of the algorithm building the partial evaluation graph. Indeed, adding or retracting one atom at the root may completely modify the shape of the resulting graph. Generalizations which were previously possible may be precluded by the retraction of just one atom, and may possibly be replaced by long branches. Conversely, long branches in the graph may be annihilated by a generalization enabled by the addition of just one atom. Similarly, the presence or absence of just one rule may yield dramatically different results. This typically cahotic behavior precludes any realistic estimation of the size of the graph (let alone the complexity of the algorithm which builds it) in terms of simple, traditional complexity factors such as the size of the abstract program call or the complexity of the methods[4].

The extension of the algorithms to handle full LinLog instead of the Simple fragment considered here would of course increase the complexity of the algorithms, but not qualitatively.

- The "why-not" modality ? is processed in a straightforward way: a goal of the form ? a (where a is an atom, in accordance with the LinLog syntax) is simply treated as a^*. However, for the construction of the partial evaluation tree, such goals should be marked, so that they can be ignored when applying the inference rules for ! and 1 (which require the context to contain only formulae prefixed with the modality ?).
- The additive connective $\oplus$ is not problematic, as it just adds an edge to an or node.
- The multiplicative connective $\otimes$ is more involved, as it requires the context to be split. It could be treated in a brute force way, by generating as many edges from an or-node as there are possible ways to split the context. A more refined strategy relies on lazy splitting. For example, the splitting of a sort ψ would result in two sorts ψ_1 and ψ_2 which are not known but only constrained by the condition $\psi = \psi_1 + \psi_2$. This technique, which has been implemented using Prolog constraint propagation, is extremely efficient.
- The connectives ! and 1 can also be treated either by the brute force method or using constraints. For example, the connective 1 only imposes the constraint that the context is empty (apart from the atoms prefixed by the why-not modality), which is simply given by the constraint $\psi = 0$.

The lazy splitting technique is of course much more efficient than the brute force one, but it has a price: when used, the detection of cycle nodes becomes more complex, as it has to deal with conditions of the form $\psi' \leq \psi$ where ψ and ψ' are only partially known. But again, that condition can be treated as a constraint itself and get involved in the global constraint resolution mechanism. We have not fully tested this solution, and, in the future, we plan to investigate the feasability of the lazy splitting technique with respect to the problem of cycle detection.

Finally, quantification, which could add some extra complexity, is irrelevant here, since, at the abstract level, we only have to deal with propositional logic. Analysis of quantification at the concrete level can be postponed to a second stage of analysis, along the lines of what is proposed in Section 4.2.

4 Refinements of the Basic Static Analysis Scheme

In order to refine our static analysis tool, we now make some elementary assumptions on the operational mechanisms involved in our "computation as proof search" hypothesis: a concrete execution can be modeled by a set of agents exploring the concrete total evaluation tree. When an agent reaches an or-node, it selects non-deterministically one outgoing edge to proceed; if there is no outgoing edge at all, the whole computation fails. At an and-node, the agent also selects an outgoing edge to proceed, but spawns copies of itself which start exploring the other outgoing edges

[4] Except in some trivial cases, as for instance when there are only two abstract atoms.

(if any); if there is no outgoing edge at all, then the agent simply terminates. A proof is successfully completed when all the agents have terminated. We make no assumptions here on how the selection of the outgoing edge at an or-node is performed (to be complete, a strategy must of course explore all the alternatives) nor what to do in case of failure.

The static analysis method presented in the previous section offers an abstract view of the computation. Indeed, each edge in the the concrete total evaluation tree corresponds to an edge in the partial evaluation graph. Thus, the evolution of the agents exploring the total evaluation tree can be traced in the partial evaluation graph.

The labels of the partial evaluation graph give, for each or-node, some information on the structure of the search agents traversing that node. Each search agent holds a concrete context characterizing the current state of the search on the branch of proof which it explores. The main information provided by the partial evaluation graph concerns the cardinality of the different atoms in this context. For example, a node labeled by a sort p, q^* reveals that the context of a search agent traversing that node contains exactly one occurrence of an atom whose abstraction is p and an indefinite number (possibly null) of occurrences of atoms whose abstraction is q (and nothing else). But other kinds of information can be inferred from the graph. The different algorithms presented in this section attach to each node in the partial evaluation graph various kinds of information which complement the basic cardinality information.

4.1 Topological Information

We first consider the results which can be inferred from a simple analysis of the topology of the partial evaluation graph. Basically, these results rely upon the notion of reachable nodes, which characterize all the possible states to which a search agent can move from a given state.

The set of (direct) successors of an or-node ν is defined as the set of or-nodes ν' such that there is an edge from ν to an and-node from which there is an edge to ν'. Reachability is defined as the transitive closure of the direct successor relation. The set $\Lambda(\nu)$ of reachable nodes from an or-node ν can be computed by a simple fix-point procedure.

Various properties can be derived using the notion of reachability. A few examples are given below.

- For any or-node ν, we take the set of accessible methods of ν to be the set of (abstract) methods which are applicable at node ν or at some node reachable from ν. This set can easily be computed, since the methods applicable at one node are explicitly contained in the graph: from each or-node, there is one outgoing edge for each applicable method.

  The set of accessible methods identifies the part of the program which the agent may need in the exploration of its own branch of proof. It thus provides a natural criterion, based on inferred behavior instead of explicit module declarations by the programmer, for partitioning the program into modules.

- Similarly, we can define the set of accessible atoms of an or-node ν as the set of (abstract) atoms which occur in the head of some accessible method of ν. Atoms which are not accessible at one node can simply be permanently ignored by any search agent traversing that node, since, by construction, these atoms cannot be involved in any future transition of that agent.

- Finally, we define "multi-instance" nodes as follows:

  Definition 6 *An or-node is said to be multi-instance if it is reachable from the output nodes of two distinct edges stemming from a common and-node.*

  By definition, a multi-instance node ν may be traversed by two agents cloned at the common predecessor and-node of the two or-nodes from which ν is reachable. Assuming that the search agents are completely independent, the two clones may possibly reach ν simultaneously. Thus, a multi-instance node is characterized by the fact that, at any time during the search, there may be more than one agent at that node. On the other hand, mono-instance nodes can be traversed by at most one agent at a time. In other words, the graph provides not only some information about the cardinality of the different components of the state of each agent, but also some information about the cardinality of the agents themselves. Figure 5 describes the algorithm for computing multi-instance nodes. Notice that the multi-instance property is stable under reachability; this could be used to improve the algorithm, since, when a node is marked as multi-instance, all the nodes reachable from it can also be immediately marked as multi-instance too, and need not be explored further.

Procedure MULTI-INSTANCE
  Foreach and-node μ in the graph **Do**
    Foreach pair of distinct outgoing edges from μ **Do**
      Let ν_1, ν_2 be the output or-nodes of these edges;
      Foreach node ν in $\Lambda(\nu_1) \cap \Lambda(\nu_2)$ **Do**
        Mark ν as multi-instance;

Figure 5: Determination of the multi-instance nodes

4.2 Structural Information

In the rest of the paper, we consider that the concrete atoms are of the form $p(a_1, \ldots, a_n)$ where p ranges over a finite set of predicates (with fixed arity) and the arguments $a_1, \ldots, a_n$ range over a possibly infinite set of values. We use the "canonical" abstract interpretation function (introduced in Section 3.2.3) which maps each atom into its predicate. In methods, the arguments of the predicates are variables, taken from an infinite set of variables, and ground instances of methods are obtained by consistently replacing variables by values.

The canonical abstract interpretation loses all the information concerning the values of the arguments in atoms, although this information is important, and especially the sharing of arguments between different atoms. Properties concerning the argument-level structure of the nodes of the partial evaluation graph can be inferred and used for various purposes. In particular, when a search agent proceeds through a cycle in the partial evaluation graph, it is interesting to analyze the evolution of a given argument of a given atom (of cardinality 1, to avoid ambiguities[5]) each time the agent traverses the same node again. The value of such an argument may, for instance, ever increase (or decrease) during the lifetime of the agent (according to some ordering depending on the type of the argument considered) and that property may be used to prove termination, or even to evaluate the complexity, of the computation being performed. For simplicity sake, we consider here only the "stability" property:

Definition 7 *Let ν be an or-node in the graph and p be an abstract atom (i.e. a predicate of arity n) of cardinality 1 at ν. The k-th argument (where $1 \leq k \leq n$) of p is said to be stable at ν if, for all possible behaviors of an agent at node ν, the value of this argument remains constant at each re-traversal of ν by that agent (strictly speaking, by any of its descendents).*

Notice that this definition is interesting only if the node ν is located on a cycle of the partial evaluation graph. Notice also that stability is essentially a temporal property: a node ν may be multi-instance and nothing relates the values of a stable argument at ν for the *different* agents which traverse ν at the *same* time; on the other hand, if the *same* agent traverses ν at two *different* times, then the values for its stable arguments remain, by definition, identical.

Determining stable arguments can be done by simulating a concrete proof search and tracing it at the abstract level in the partial evaluation graph, comparing the successive states of a search agent at each retraversal of a node. Figure 6 describes the algorithm for computing stable arguments at each node. It makes use of the following terminology:

- A *pattern of instantiation* is a multiset of atoms with pairwise distinct predicates and whose arguments are variables. A pattern of instantiation is used to encode the part of an agent state concerning only the predicates of cardinality 1 at some node.
- A *history* is a list of pairs $\langle \nu; \sigma \rangle$ where ν is a node and σ is a pattern of instantiation for the predicates of cardinality 1 at node ν. A history encodes the successive states of an agent during its traversal of a path in the graph.

When a node is reached, the algorithm first tests whether the current state is an instance of a previous one at the same node in the history: in this case, the exploration of the current path is terminated, since exploring it further would yield the same (or more specific) sequences of states as those obtained from the ancestor state, generating no new unstable arguments; otherwise, the arguments in the current state are compared with their values in previous states at the same node (if any) and marked unstable if different; the exploration of the path then continues. At the end of the algorithm, the stable arguments at each node, are those arguments which have not been marked unstable.

[5] The technique presented here could also apply to atoms of cardinality other than 1, but it is more difficult in that case to identify clearly the atom to which the tracked argument refers.

Procedure ARGUMENT-LEVEL-ANALYSIS
  Let ν_o be the root node;
  Let σ_o be the most general pattern of instantiation;
  Let κ_o be the empty history;
  Call PROPAGATE($\nu_o, \sigma_o, \kappa_o$);

Procedure PROPAGATE(ν: node, σ: pattern of instantiation, κ: history)
  Let $\sigma' \leftarrow$ FILTER(ν, σ);
  If $\neg$INSTANCE(σ', ν, κ) **Then**
    Call MARK-UNSTABLE(σ', ν, κ)
    Let κ' be the history composed of the pair $\langle \nu, \sigma' \rangle$ appended to κ;
    Foreach outgoing edge from ν to an and-node μ **Do**
      Let M be the method attached to that edge;
      Let $[\sigma_1, \ldots, \sigma_n], \theta \leftarrow$ APPLY(M, σ')
      For $i = 1, \ldots, n$ **Do**
        Let ν_i be the i^{th} outgoing edge from μ
        Call PROPAGATE($\nu_i, \theta.\sigma_i, \theta.\kappa'$);

Procedure MARK-UNSTABLE(σ: pattern of instantiation, ν: node, κ: history)
  Foreach predicate p (of arity n) of cardinality 1 at ν **Do**
    For $k = 1, \ldots, n$ **Do**
      Foreach pair $\langle \nu; \sigma' \rangle$ in κ **Do**
        If the k^{th} argument of p has a different value in σ and σ' **Then**
          Mark the k^{th} argument of p at ν as unstable;

Figure 6: Determination of the stable arguments

The "most general pattern of instantiation" σ_o, used at the initialization of the algorithm, denotes the pattern which contains one occurrence of each predicate and such that all their arguments are pairwise distinct variables. Three side procedures are used in this algorithm:

- Procedure FILTER(ν, σ) returns, for a node ν and a pattern of instantiation σ, the part of σ which concerns only the predicates of cardinality 1 at ν.
- Procedure APPLY(M, σ), where M is a method and σ a pattern of instantiation, has the following behavior: (*i*) it renames the variables in M so that it does not share any variable with σ; (*ii*) it deletes from the head of M those atoms which have no counterparts in σ (this may happen since σ contain only atoms of cardinality 1); (*iii*) it unifies the remaining atoms of the head of M with the corresponding atoms in σ which it removes from σ; let θ be the most general unifying substitution (which is only a renaming of variables); (*iv*) it decomposes the body of M in the context of what remains of σ; (*v*) it returns the substitution θ together with the (possibly empty) list of patterns of instantiation obtained at the end of the decomposition. For example, let's consider

$$\begin{array}{lcl} \sigma & = & p(U), q(V), r(U,W) \\ M & = & p(X) \parr q(X) \parr c(X,Y) \circ\!\!\perp s(X) \,\&\, t(Y) \end{array}$$

  The head of M is first reduced to $p(X) \parr q(X)$ (the atom $c(X,Y)$ does not occur in σ) then unified with the subset $p(U), q(V)$ of σ using the substitution $\theta = \{X/U, X/V\}$. The decomposition of the body of M then yields two patterns of instantiation $\sigma_1 = r(U,W), s(X)$ and $\sigma_2 = r(U,W), t(Y)$. The returned result is the pair $[\sigma_1, \sigma_2], \theta$.
- Procedure INSTANCE(σ, ν, κ), where σ is a pattern of instantiation of the predicates of cardinality 1 at node ν and κ is a history, tests whether there exists in κ a pair of the form $\langle \nu; \sigma' \rangle$ such that σ is a variant or an instance of σ' (i.e. there is a mapping θ, from variables to variables, such that $\sigma = \theta\sigma'$).

Theorem 10 *The algorithm* ARGUMENT-LEVEL-ANALYSIS *(Figure 6) always terminates.*

Demonstration: The algorithm is ensured to terminate, because there is only a finite number of patterns of instantiation (modulo renaming of variables) and hence, the exploration of a looping path always returns to a pattern of instantiation which is a variant or an instance of a previous pattern. □

Many properties other than stability could be investigated using a similar technique. For example, if we now allow the use of first order terms as arguments of atoms, we can perform some type inference analysis by tracing, when possible, the term skeletons of each arguments, up to a certain depth so as to ensure termination.

5 Applications

The results presented in Section 3 are very general and can be applied to any kind of analysis of proofs in Linear Logic (e.g. decision problems for various fragments of Linear Logic, as in [38]). The results of Section 4 are more focused on proof search processes, though it makes minimal assumptions. As far as programming is concerned, our static analysis tool applies quite naturally to compile time optimizations in programming paradigms where computation can be represented by proof search in Linear Logic.

5.1 The Concurrent Language LO

On the side of language optimizations, one main application of the techniques described in the previous sections has been in the compilation and execution of the concurrent language LO [7]. The operational semantics of LO is essentially captured in terms of the proof search in the Linear Logic fragment Simple LinLog we have assumed in the previous sections. LO is based on two main search mechanisms: (*i*) expansion of the open leaves of the proof tree and (*ii*) adjunction of atoms simultaneously at all the nodes of the tree.

- The first mechanism accounts for agents evolution in a multi-agent system (creation, transformation, termination). Agents are viewed as reactive entities which execute concurrently and are themselves internally composed of concurrent threads.
- The second mechanism, explicitly controlled by a special marker (`^`) in the methods, accounts for associative communication among agents. Communication is basically viewed here as broadcast, although no assumption is made on its implementation, which can be based on a point-to-point, address based scheme.

On the side of efficiency, there are two main problems in executing LO programs:

Non-determinism: in the selection of the method in the LinLog inference figure.

Saturation: following from the fact that, with broadcasting, each agent must receive and store all the broadcast messages, even those for which it has no use.

The static analysis technique presented in the previous section can be exploited to alleviate such problems. Indeed, it has been shown here how the partial evaluation graph can be used to detect methods which will never be able to match the state of certain agents, as well as detect items which, in a given agent, will never be consumed by any method; in this way, both the amount of non-determinism and the amount of saturation can be decreased at compile-time. Notice here that stable arguments may play a crucial role. Indeed, for example, an agent A may hold an argument which can be used as an "address" for specific communication with A. To communicate specifically with A, the sender just puts the address inside the message (at an agreed position), and, although the message is broadcast, only the agent A, which is acquainted with this address, can process it. Typically, such address arguments are stable (an agent does not want to change its address during its lifetime), and it is useful to detect them in advance, so as to eventually implement, for that agent, the general broadcast communication mechanism using a specific point to point communication mechanism.

5.1.1 A Sample Program

Figure 7 presents a chart-parser written in LO (a variant of this program is explained in details in [7]). This program is intended to run from within ForumTalk [3], a distributed programming environment designed on the concepts of LO, where several agents have access to the forum and can broadcast messages. The `extern` declaration at the begining of the program characterizes atoms which can be broadcast from agents outside the program. In the static analysis, without any further information on such atoms, we assume that their cardinality is indefinite. In fact, all the broadcast predicates (whether generated by the program or from the outside) are treated in this way. Furthermore, we assume that the proof search is initialized by a sequent containing a single atom which is an instance of an `entry` predicate as declared at the begining of the program. Hence, the initial program call for the static analysis is of the form $e, \{m^*\}_{m \in \mathcal{M}}$ where e is the entry predicate and $\mathcal{M}$ is the set of broadcast predicates (both internal and external).

```
extern parse/3, answer/2, extract/4.
entry main/2.
main(G,D) <>- grammar(G) & dico(D) & scanner.

% Evolution of the incomplete trees
itree(M,C,T) @ ctree(M,N,C,T1) <>- itree(M,C,T) & new(N,T-T1).
new(N,T) @ tail([C|Cs]) @ ^seek(N,C) <>- itree(N,C,T) @ tail(Cs).
new(N,T) @ tail([]) @ wait(M,C) @ ^ctree(M,N,C,T) <>- #t.

% The grammar
grammar([r(C,Cs)|G]) <>- head(C) @ tail(Cs) & grammar(G).
grammar([]) <>- #t.
head(C) @ seek(N,C) @ pos(N) <>- head(C) & new(N,C) @ wait(N,C).

% The dico
dico(D) @ word(W,M,N) @ extract(D,W,Cs,D1) <>-
    dico(D1) & w(W) @ entries(Cs) @ word(W,M,N).
entries([C|Cs]) <>- entry(C) @ entries(Cs).
entries([]) @ w(W) <>- wdico(W).
wdico(W) @ word(W,M,N) <>- wdico(W) & gctree(W,M,N).
gctree(W,M,N) @ entry(C) @ ^ctree(M,N,C,C-W) <>- gctree(W,M,N).

% The scanner
scanner @ parse(Q,S,C) @ ^seek(M,C) <>-
    scan(S,M) @ target(Q,M,C) & scanner.
scan([W|S],M) @ ^pos(M) @ ^word(W,M,N) <>- scan(S,N).
scan([],N) @ target(Q,M,C) <>- wait(Q,M,N,C).
wait(Q,M,N,C) @ ctree(M,N,C,T) @ ^answer(Q,T) <>- wait(Q,M,N,C).
```

Figure 7: A chart-parser in LO

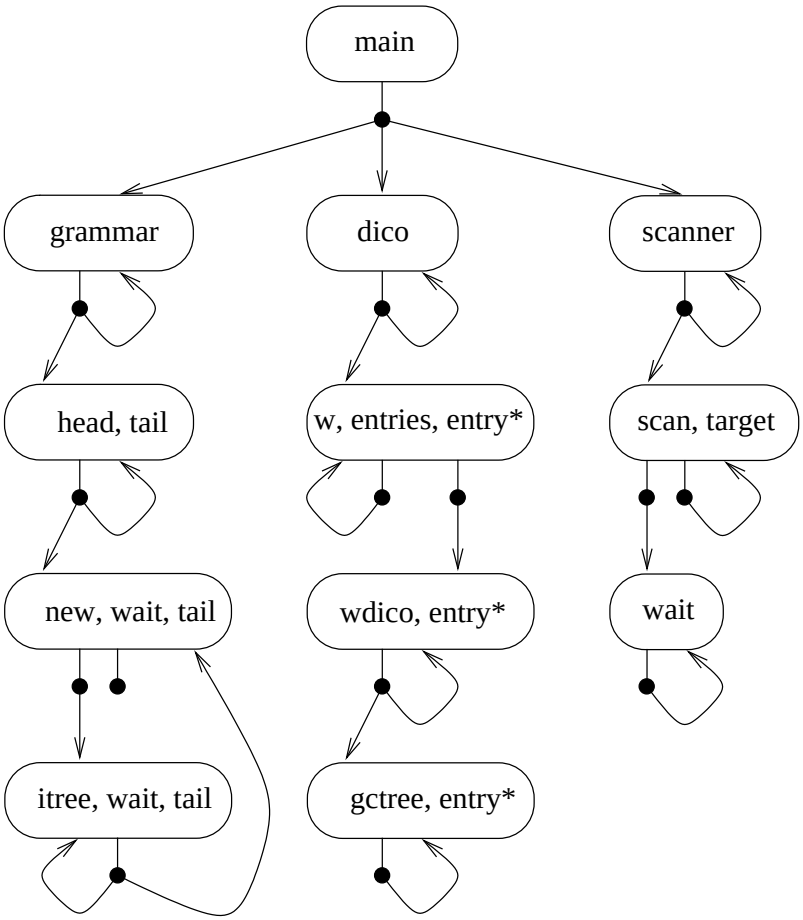

Figure 8: The partial evaluation graph for the chart-parser

case	Nb agents	Static Analysis **Off**		Static Analysis **On**	
		time	size	time	size
1	57	< 1	4749	< 1	2279
2	77	1.5	10012	0.4	5019
3	111	4.5	24559	0.9	13313
4	187	20	82298	3.5	48954
5	395	230	417327	28	265980

case	Sentence length	Nb parse trees
1	7	2
2	10	5
3	13	15
4	16	42
5	19	132

Figure 9: Measurement for the sample LO program: times are CPU times in seconds, measured on a Sun Sparc station; in each case, the size represents the cumulated number of components of the agents

Figure 8 shows the partial evaluation graph for the chart parser. The broadcast predicates have been omitted, as they occur at all the nodes. The graph makes clear the structure of the program, consisting of three modules (scanning, parsing, and dictionary access). Analysis of stable arguments shows, for example, that the argument of `head` at the node labeled `head,tail` is stable (a grammar rule never changes its head, nor its tail) so that any `seek` request reaching a rule can be immediately discarded if it does not match the head of the rule.

5.1.2 Performance Measurements

The static analysis algorithms presented in this paper have all been implemented in the ForumTalk platform [3]. However, only the results of the basic static analysis (Section 3) have been integrated to the LO compiler of ForumTalk, not those of Section 4. In spite of this, we could already measure important performance gains.

For example, the tables of Fig. 9 give measurements for different runs of the chart parser (above). Each case corresponds to the parsing of a sentence. The length (in words) of each sentence and the corresponding number of generated parse trees for the full sentence is shown in the lower table of Fig. 9. The sentences have been artificially selected so as to have as many ambiguities as possible and thus generate many parse trees. The upper table of Fig. 9 shows the number of agents generated in each case (most of them correspond to incomplete parse trees) and give the measurements themselves. They concern, on the one hand, the overall cpu time between the submission of a parse request and the last reply, and, on the other hand, the cumulated number of atoms in the agents. They characterize quite precisely the two effects (resp. non determinism for the time measurement and saturation for the size measurement) mentioned in the previous section. Most of the atoms in an agent are the messages which this agent has received (via the broadcast mechanism), and which, in the case of static analysis, have not been discarded by the garbage collection mechanism.

As far as saturation is concerned, the garbage collection of unused atoms allowed by static analysis discards about half the total number of atoms. This already interesting result would be considerably improved if the refined information generated by the second step of static analysis (Section 4) had been integrated to the LO compiler. Thus, we estimate that, using stable arguments information, the number of garbage collected atoms would reach 90%! Concerning time measurement, the effect of static analysis on time performances is considerable: up to a factor of 8 (in the last case). Part of the improvement comes from the reduction of the number of messages per agent, which reduces the amount of matching operations, but this is probably not the most significant cause, since, here, only half the messages are garbage collected. More important is the reduction of the non-determinism effect due to the elimination of many provably unapplicable rules at each agent.

The sample application discussed here is, obviously, a toy application. However, it incorporates many patterns of coordination which we have found in other, more realistic applications which we have developed, for example the Knowledge Brokers [4]. In this application, the time performance of the core of the system, written in ForumTalk, is not immediately critical, in the sense that it is, initially, negligeable w.r.t. the network performance (the application runs on the World Wide Web). However, static analysis may still be very useful, to avoid a degradation of the performance over a long period, due to the accumulation of useless messages. It might not be realistic to rely only on the static analysis tool to remove all the potential sources of accumulation of useless messages. However, once a source is identified, it is easy for the programmer to add a method to the program which performs explicitly the requested garbage collection. In fact, the static analysis tool should be seen as an aid to the programmer with which to interact during the development of a program, in order to improve its performances. From that point of

view, it is improtant to be able to graphically display the information inferred by static analysis, in particular the partial evaluation graph.

5.2 Other Applications

Let's stress here that the application of static analysis has been fully investigated mainly for the language LO. However, due to the generality of our approach (we focus, at least initially, on proofs rather than proof search), it can easily be applied to other frameworks. Thus, all the systems mentioned in the introduction [35, 50, 21, 42, 14, 49, 38] rely in some way or another on proof search in Linear Logic and are therefore amenable to the kind of static analysis we have presented here. In most cases, once converted to the LinLog normal form, the formulae used in these systems belong to the Simple LinLog fragment (and even, often, the fragment without &). Notice here that the use of the connective $\otimes$ does not *per se* means that we are outside the Simple LinLog fragment. For example, a formula $a \otimes b \perp\!\!\!\circ c \otimes d$ (as can be found, e.g., in the Linear version of CCP [50]) can be rewritten, by transposition, $a^{\perp} \parr b^{\perp} \circ\!\!\!\perp c^{\perp} \parr d^{\perp}$, and, if, by proceeding in this way on all the formulae, all the occurrences of the atoms a, b, c, d are replaced by their duals, we then are in the Simple LinLog fragment (up to an irrelevant substitution of the atoms by their duals). The same kind of transformation to Simple LinLog formulae applies quite straightforwardly to ACL [35]. Anyway, it has been shown in Section 3.4 that our static analysis technic, mainly focused here on Simple LinLog, could be extended to full LinLog.

Furthermore, the LinLog approach to encoding reactive concurrent computations has been quite effective in amalgamating and declaratively reconstructing a number of different approaches to concurrency which are "a-priori" not based on proof search and Linear Logic: in general, all shared data-space languages can be reconstructed in this way (Linda [27], the Cham [10], Gamma [8], production systems [11, 47], Maude [44]). In these frameworks, our static analysis techniques could be used to discard pieces of code which are never used with respect to certain initial states: they correspond to the unaccessible methods or atoms determined by the analysis. A similar technique has been recently applied in the context of class-based object-oriented languages for compile-time detection of dead code [46]. Of course, the difficulty in all these cases is to find the appropriate abstraction mapping, or, if the canonical one is to be used, to select the right set of predicates to represent the different systems in LinLog.

6 Related Work

Abstract interpretation has been formalized in [19, 18] in the framework of imperative languages, and has later been adopted for the analysis of functional and logic programming languages.

There is a large body of work in static analysis of Logic Programming based on abstract interpretation and partial evaluation [9, 15, 16, 17, 41, 40, 43]. Some of the techniques used here, such as the encoding of computations via an and-or tree [12], or the approximation of these computations using lattice theoretic notions, are common to these approaches, especially those based on top-down analysis [23, 24, 33]. However, the work presented here differs from analogous work in traditional Logic Programming from two main perspectives: (*i*) the core technique presented in this paper does not rely on any assumption concerning the operational semantics of proof search; indeed, we are interested in the resource manipulation aspects of computations, which are intrinsic to Linear Logic, and not derived from explicit ad-hoc search mechanisms as in traditional Logic Programming frameworks; on the other hand, we are not concerned with more model theoretic aspects (Herbrand models, satisfiability) deriving from the traditional view of Logic Programming in Classical Logic; (*ii*) we wish to make as little assumptions as possible on the nature of the resources manipulated by the agents in the computation; hence, our static analysis tool does not assume the use of first-order terms nor, consequently, the use of unification and answer substitutions, one of the crucial mechanisms in traditional Logic Programming.

The techniques which have been developed to handle first-order terms and instantiation propagation could be integrated to our framework, since they amount to the building of a similar kind of and-or tree. But this paper shows that, in Linear Logic, we can already infer many useful informations without having to enter the structure of first order terms. Notice that Constraint Logic Programming also tends to discard the notion of first-order terms as sole data-structure, and replace it by the notion of sets of constraints; static analysis techniques have also been explored in this framework [28, 32]. The notion of constraint is close to our notion of resource; however, constraints are not considered bounded resources (they can be re-used as many times as needed in different constraint propagation processes), thus preventing any form of cardinality analysis such as the one presented in this paper.

A similar kind of cardinality analysis, has also been developed for ACL [34], where a combination of type-inference and so-called "effect" inference is proposed: the effect of a process expression (in ACL) characterizes the number of read/write operations performed on the channels it involves, which can, to a certain extent, be anticipated by analysis. This analysis is specific to the computational model of ACL (which, incidently, is also related to Linear Logic proof search), whereas, as explained above, a large part of our analysis (section 3) is independent of any computational use of the proofs and applies to the proofs themselves. As for the part which

is dependent on a specific operational interpretation (section 4), it focuses on the cardinality of resources rather than that of the read/write operations.

7 Future work

There are basically two main directions in which the work presented in this paper can be extended.

1. One is to keep the basic structure of the partial evaluation graph as it has been presented here, and to provide a unified framework in which the different properties which are inferred and attached at each node can be expressed and manipulated. This would mean to devise (i) a suitable linguistic framework allowing the expression of constraints on various aspects of the structure of the search agents at each node, and also on the nodes themselves (e.g. topological information), and (ii) a general mechanism for propagating these constraints along the edges of the graph. Such a general linguistic framework has been proposed in the context of Unity [20], based on temporal logic inferences [39, 37].

2. Another direction consists of modifying the way in which the graph is built so as to incorporate, at the time of the graph construction itself, more refined information than the mere cardinality information provided by sorts. That would require to define a more complex lattice than the lattice of sorts, and to adapt consequently the generalization step which ensures termination of the graph construction.

   Notice that in the current framework, generalization occurs when the sort at one node *totally* covers the sort at one of its predecessor. Considering some form of *partial* covering could lead to a more compact representation of the graph, better suited to the analysis of internal parallelism inside agents.

On the application side, apart from the compilation optimizations which have been hinted at in section 5, the partial evaluation graph could provide the support for a visual representation of the execution (similar to Petri nets or State Charts [30]) and could be used for tracing or debugging purposes.

A Proof of Theorem 6

We give here a proof of theorem 6 on which the convergence of our method relies. As $\mathcal{A}$ is finite, we may assume, without loss of generality, that $\mathcal{A} = \{1, \ldots, k\}$ for some integer k. So we have to show, for all integer k, the property $\mathcal{P}(k)$:

> For all infinite sequence $(\vec{x}_n)_{n\in\mathbf{N}}$ of elements of $\bar{\mathbf{N}}^k$, there exists indices p, q such that $p < q$ and $\vec{x}_p \preceq \vec{x}_q$.

Demonstration: This property is shown by induction.

- $\mathcal{P}(1)$ is obvious: indeed, in this case, each $\vec{x}_n$ is an integer (or ∞) and p can be chosen as the index of the smallest $\vec{x}_n$ (which always exists since any set in $\bar{\mathbf{N}}$ has a smallest element); then, we can take $q = p + 1$.

- Now, let us assume that $\mathcal{P}(k)$ holds, and let's prove $\mathcal{P}(k+1)$. Let $(\vec{x}_n)_{n\in\mathbf{N}}$ be an infinite sequence of elements of $\bar{\mathbf{N}}^{k+1}$ and let's consider the sequence $(z_n)_{n\in\mathbf{N}}$ of elements of $\bar{\mathbf{N}}$ defined by:

  $$z_n \;=\; \vec{x}_n[k+1]$$

  Any sequence in $\bar{\mathbf{N}}$ has a non decreasing subsequence; therefore there exists a monotonic function $\sigma : \mathbf{N} \mapsto \mathbf{N}$ such that for all $m, n \in \mathbf{N}$ if $m < n$ then $\sigma(m) < \sigma(n)$ and $z_{\sigma(m)} \leq z_{\sigma(n)}$.

  Let's now consider the sequence $(\vec{y}_n)_{n\in\mathbf{N}}$ of elements of $\bar{\mathbf{N}}^k$ defined by:

  $$\forall j \in \{1, \ldots, k\} \quad \vec{y}_n[j] \;=\; \vec{x}_{\sigma(n)}[j]$$

  By the induction hypothesis $\mathcal{P}(k)$, there exists indices p, q such that $p < q$ and $\vec{y}_p \preceq \vec{y}_q$. Hence

  - For all $j = 1, \ldots, k$, we have, since $\vec{y}_p \preceq \vec{y}_q$

    $$\vec{x}_{\sigma(p)}[j] = \vec{y}_p[j] \leq \vec{y}_q[j] = \vec{x}_{\sigma(q)}[j]$$

  - For $j = k + 1$, we have, since $p < q$ and $\sigma(p) < \sigma(q)$,

    $$\vec{x}_{\sigma(p)}[k+1] = z_{\sigma(p)} \leq z_{\sigma(q)} = \vec{x}_{\sigma(q)}[k+1]$$

Let $p' = \sigma(p)$ and $q' = \sigma(q)$. We have

$$p' < q' \text{ and } \forall j = 1, \ldots, k+1, \quad \vec{x}_{p'}[j] \leq \vec{x}_{q'}[j]$$

Hence $\vec{x}_{p'} \preceq \vec{x}_{q'}$. Therefore, $\mathcal{P}(k+1)$ holds.

By induction, $\mathcal{P}(k)$ holds for all k. □

References

[1] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1–2):3–57, 1993.

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

[3] J-M. Andreoli. Coordination in lo. In J-M. Andreoli, C. Hankin, and D. Le Metayer, editors, *Coordination Programming: Mechanisms, Models and Semantics*. Imperial College Press, 1996.

[4] J-M. Andreoli, U. Borghoff, and R. Pareschi. Constraint based knowledge brokers. In *Proc. of PASCO'94*, Linz, Austria, 1994.

[5] J-M. Andreoli, P. Ciancarini, and R. Pareschi. Interaction abstract machines. In G. Agha, A. Yonezawa, and P. Wegner, editors, *Research Directions in Concurrent Object Oriented Programming*, pages 257–280. MIT Press, Cambridge, Ma, U.S.A., 1993.

[6] J-M. Andreoli and R. Pareschi. Logic programming with sequent systems: a linear logic approach. In *Proc. of the Workshop on Extensions of Logic Programming*, Lecture Notes in Artificial Intelligence 475, (Springer Verlag), Tübingen, Germany, 1990.

[7] J-M. Andreoli and R. Pareschi. Communication as fair distribution of knowledge. In *Proc. of OOPSLA'91*, Phoenix, Az, U.S.A., 1991.

[8] J-P. Banâtre, A. Coutant, and D. Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4(2):133–145, 1988.

[9] R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics based bottom-up abstract interpretation of logic programs. *IEEE Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.

[10] G. Berry and G. Boudol. The chemical abstract machine. In *Proc. of the 17th ACM Symposium on Principles of Programming Languages*, San Francisco, Ca, U.S.A., 1990.

[11] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS-5*. Addison-Wesley, Reading, Ma, U.S.A., 1985.

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

[13] N. Carriero and D. Gelernter. *How to Write Parallel Programs*. MIT Press, Cambridge, Ma, U.S.A., 1990.

[14] S. Cerrito. A linear semantics for allowed logic programs. In *Proc. of the 5th IEEE Symposium on Logic in Computer Science*, Philadelphia, Pa, U.S.A., 1990.

[15] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Theoretical Computer Science*, 124, 1994.

[16] P. Codognet and G. Filé. Computations, abstractions and constraints in logic programs. In *Proc. of the 4th International Conference on Programming Languages (ICCL)*, Oakland, Ca, U.S.A, 1992.

[17] M. Corsini and G. Filé. A complete framework for the abstract interpretation of logic programs: Theory and application. Technical report, Università di Padova, Padova, Italy, 1989.

[18] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoint. In *Proc. of the 4th ACM symposium on Principles Of Programming Languages*, 1977.

[19] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of the 6th ACM symposium on Principles Of Programming Languages*, 1979.

[20] H.C. Cunningham and G.-C. Roman. A UNITY-style programming logic for shared dataspace programs. *IEEE Transaction on Parallel and Distributed Systems*, 1(3), 1990.

[21] M. Dalrymple, A. Hinrichs, J. Lamping, and V. Saraswat. The resource logic of complex predicate interpretation. In *Proc. of the 1993 Republic of China Computational Linguistics Conference (ROCLING)*, Hsitou National Park, Taiwan, 1993.

[22] M. Dalrymple, J. Lamping, Pereira F.C.N., and V. Saraswat. Linear logic for meaning assembly. In *Proc. of CLNLP'95*, Edinburgh, U.K., 1995.

[23] S. Debray. Efficient dataflow analysis of logic programs. In *Proc. of POPL'88*, San Diego, Ca, U.S.A., 1988.

[24] S. Debray and D.S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5:207–230, 1988.

[25] C. Fouquere and J. Vauzeilles. Inheritance with exception. In J-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*. Cambridge University Press, Cambridge, U.K., 1995.

[26] D. Galmiche and G. Perrier. On proof normalization in linear logic. *Theoretical Computer Science*, 135:67–110, 1994.

[27] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[28] R. Giacobazzi, S. Debray, and G. Levi. Generalized semantics and abstract interpretation for constraint logic programming. Technical report, Università di Pisa, Pisa, Italy, 1992.

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

[30] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3), 1987.

[31] J. Harland and D. Pym. The uniform proof-theoretic foundation of linear logic programming. In *Proc. of the International Logic Programming Symposium*, San-Diego, Ca, U.S.A., 1991.

[32] G. Janssens and M. Bruynooghe. Towards a framework for abstract interpretation of constraint logic programming. In *Proc. of META'92*, Uppsala, Sweden, 1992.

[33] N.D. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood, Chichester, U.K., 1987.

[34] N. Kobayashi, M. Nakade, and A. Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *Proc. of SAS'95*, 1995.

[35] N. Kobayashi and A. Yonezawa. Acl — a concurrent linear logic programming paradigm. In *Proc. of ILPS'93*, Vancouver, B.C., Canada, 1993.

[36] N. Kobayashi and A. Yonezawa. Asynchronous communication models based on linear logic. *Formal Aspects of Computing*, 7(2):113–149, 1995.

[37] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[38] P. Lincoln, J. Mitchell, A. Scedrov, and N. Shankar. Decision problems for propositional linear logic. *Annals of Pure and Applied Logic*, 56:239–311, 1992.

[39] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer Verlag, 1992.

[40] K. Marriot and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In *Proc. of ICLP'88*, Seattle, Wa, U.S.A., 1988.

[41] K. Marriot and H. Søndergaard. Semantic based dataflow analysis of logic programs. In *Proc. of the IFIP 11th World Computer Congress*, San Francisco, Ca, U.S.A, 1989.

[42] M. Masseron, C. Tollu, and J. Vauzeille. Generating plans in linear logic. *Theoretical Computer Science*, 113:349–370, 1993.

[43] C. Mellish. Abstract interpretation of prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 181–198. Ellis Horwood, Chichester, U.K., 1987.

[44] J. Meseguer. A logical theory of concurrent objects and its realization in the MAUDE language. In G. Agha, A. Yonezawa, and P. Wegner, editors, *Research Directions in Concurrent Object Oriented Programming*. MIT Press, Cambridge, Ma, U.S.A., 1992.

[45] G. Mints. Resolution calculus for first-order linear logic, 1991. Unpublished manuscript.

[46] N. Oxhøj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In *Proc. of ECOOP'92*, Utrecht, The Netherlands, 1992.

[47] H. Penny Nii. Blackboard systems. In A. Barr, Cohen P., and Feigenbaum E., editors, *The Handbook of Artificial Intelligence, vol. 4*, pages 1–82. Addison-Wesley, Reading, Ma, U.S.A., 1989.

[48] J-L. Peterson. *Petri-Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, NJ, U.S.A., 1981.

[49] U. Reddy. A typed foundation for directional logic programming. In *Proc. of the workshop on the Extensions of Logic Programming*, Bologna, Italy, 1992.

[50] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, Pittsburg, Pa, U.S.A., 1989.