GUEST TALKS

Observations about using logic as a specification language

Dale Miller *
200 S. 33rd Street, Computer Science Department
University of Pennsylvania, Philadelphia, PA 19104–6389 USA
dale@saul.cis.upenn.edu
Phone: +1-215-898-1593, Fax: +1-215-898-0587

Abstract

This extended abstract contains some non-technical observations about the roles that logic can play in the specification of computational systems. In particular, computation-as-deduction, meta-programming, and higher-order abstract syntax are briefly discussed.

1 Two approaches to specifications

In the specification of computational systems, logics are generally used in one of two approaches. In one approach, computations are mathematical structures, containing such items as nodes, transitions, and state, and logic is used in an external sense to make statements *about* those structures. That is, computations are used as models for logical expressions. Intensional operators, such as the modals of temporal and dynamic logics or the triples of Hoare logic, are often employed to express propositions about the change in state. For example, next-time modal operators are used to describe the possible evolution of state; expressions in the Hennessey-Milner are evaluated against the transitions made by a process; and Hoare logic uses formulas to express pre- and post-conditions on a computation's state. We shall refer to this approach to using logic as *computation-as-model.* In such approaches, the fact that some identifier x has value 5 is represented as, say a pair $\langle x, 5\rangle$, within some larger mathematical structure, and logic is used to express propositions *about* such pairs: for example, $x > 3 \wedge x < 10$.

A second approach uses logical deduction to model computation. In this approach the fact that the identifier x has value 5 can be encoded as the proposition

*The work reported here has been funded in part by ONR N00014-93-1-1324, NSF CCR-91-02753, and NSF CCR-92-09224.

"x has value 5." Changes in state can then be modeled by changes in propositions within a derivation. Of course, changing state may require that a proposition no longer holds while a proposition that did not hold (such as "x has value 6") may hold in a new state. It is a common observation that such changes are naturally supported by linear logic and that deduction (in particular, backchaining in the sense of logic programming) can encode the evolution of a computation. As a result, it is possible to see the state of a computation as a logical formula and transitions between states as steps in the construction of a proof. We shall refer to this approach to using logic as *computation-as-deduction.*

There are many ways to contrast these two approaches to specification using logic. For example, consider their different approaches to the "frame problem." Assume that we are given a computation state described as a model, say M_1, in which it is encoded that the identifier x is bound to value 5. If we want to increment the value of x, we may need to characterize all those models M_2 in which x has value 6 and *nothing else has changed.* Specifying the precise formal meaning of this last clause is difficult computationally and conceptually. On the other hand, when derivations are used to represent computations directly, the frame problem is not solved but simply avoided: for example, backchaining over the clause

$$x \text{ has value } n \multimap x \text{ has value } n+1$$

might simply change the representation of state in the required fashion.

In the first approach to specification, there is a great deal of richness available for modeling computation, since, in principle, such disciplines as set theory, category theory, functional analysis, algebras, *etc.*, can be employed. This approach has had, of course, a great deal of success within the theory of computation.

In contrast, the second approach seems thin and feeble: the syntax of logical formulas and proofs contains only the most simple structures for representing computational state. What this approach lacks in expressiveness, however, is ameliorated by the fact that it is more intimately connected to computation. Deductions, for example, seldom make reference to infinity (something commonly done in the other approach) and steps within the construction of proofs are generally simple and effective computations. Recent developments in proof theory and logic programming have also provided us with logics that are surprisingly flexible and rich in their expressiveness. In particular, linear logic [6] provides flexible ways to model state, state transitions, and some simple concurrency primitives, and higher-order quantification over typed λ-terms provides for flexible notions of abstraction and encodings of object-level languages. Also, since specifications are written using logical formulas, specifications can be subjected to rich forms of analysis and transformations.

To design logics (or presentations of logics) for use in the computation-as-deduction setting, it has proved useful to provide a direct and natural operational interpretation of logical connective. To this end, the formalization of *goal-directed search* using *uniform proofs* [14, 16] associates a fixed, "search semantics" to logical

connectives. When restricting to uniform proofs does not cause a loss of completeness, logical connectives can be interpreted as fixed search primitives. In this way, specifier can write declarative specifications that map directly to descriptions of computations. This analysis of goal-directed proof search has lead to the design of the logic programming languages λProlog, Lolli, LO, and Forum. Some simple examples with using these languages for specifications can be found in [1, 10, 14]. The recent thesis [2] provides two modest-sized Forum specifications: one being the operational semantics of a functional programming language containing references, exceptions, and continuation passing, and the other being a specification of a pipe-lined, RISC processor.

> *Observation 1.* Logic can be used to make statements about computation by encoding states and transitions directly using formulas and proof. This use of logic fits naturally in a logic programming setting where backchaining can denote state transition. Both linear logic and higher-order quantification can add greatly to the expressiveness of this paradigm.

2 An example

The following specification of reversing a list and the proof of its symmetry illustrates how the expressiveness of higher-order linear logic can provide for natural specifications and convenient forms of reasoning.

any number of times. $\multimapinv$ only once $\multimap$

```
reverse L K :- pi rv\(
  pi X\(pi M\(pi N\(rv (X::M) N :- rv M (X::N)))) => rv nil K -:
   rv L nil).
```

Here we use a variant of λProlog syntax: in particular, lists are constructed from the infix `::` and `nil`; `pi X\` denotes universal quantification of the variable `X`; `=>` denotes intuitionistic implication; and, `-:` and `:-` denote linear implication and its converse. This one example combines some elements of both linear logic and higher-order quantification.

To illustrate this specification, consider proving the query

```
?- reverse (a::b::c::nil) Q.
```

Backchaining on the definition of reverse above yields a goal universally quantified by `pi rv\`. Proving such a goal can be done by instantiating that quantifier with a new constant, say `rev`, and proving the result, namely, the goal

```
  pi X\(pi M\(pi N\(rev (X::M) N :- rev M (X::N)))) => rev nil Q -:
   rev (a::b::c::nil) nil).
```

Thus, an attempt will be made to prove the goal `(rev (a::b::c::nil) nil)` from the two clauses

$w \longrightarrow \lambda x \lambda y ((w\ y x)^{\perp})$

```
pi X\(pi M\(pi N\(rev (X::M) N :- rev M (X::N)))).
rev nil Q.
```

(Note that the variable Q in the last clause is free and not implicitly universally quantified.) Given the use of intuitionistic and linear implications, the first of these clauses can be used any number of times while the second must be used once (natural characterizations of inductive and initial cases for this example). Backchaining now leads to the following progression of goals:

```
rev (a::b::c::nil)  nil.
rev (b::c::nil) (a::nil).
rev (c::nil) (b::a::nil).
rev  nil   (c::b::a::nil).
```

and the last goal will be proved by backchaining against the initial clause and binding Q with (c::b::a::nil).

It is clear from this specification of reverse that it is a symmetric relation: the informal proof simply notes that if the table of rev goals above is flipped horizontally and vertically, the result is the core of a computation of the symmetric version of reverse. Given the expressiveness of this logic, the formal proof of this fact directly incorporates this main idea.

Proposition. Let l and k be two lists and let $\mathcal{P}$ be a collection of clauses in which the only clause that contains an occurrence of reverse in its head is the one displayed above. If the goal (reverse l k) is provable from $\mathcal{P}$ then the goal (reverse k l) is provable from $\mathcal{P}$.

Proof. Assume that the goal (reverse l k) is provable from $\mathcal{P}$. Given the restriction on occurrences of reverse in $\mathcal{P}$, this goal is provable if and only if it is proved by backchaining with the above clause for reverse. Thus, the goal

$\vdash \forall r \sigma$

```
pi rv\(
  pi X\(pi M\(pi N\(rv (X::M) N :- rv M (X::N)))) =>
  rv nil k -: rv l nil)
```

is provable from $\mathcal{P}$. Since this universally quantified formula is provable, any instance of it is provable. Let rev be a new constant not free in $\mathcal{P}$ of the same type as the variable rv. The formula that results from instantiating this quantified goal with the λ-term x\y\(not (rev y x)) (where \ is the infix symbol for λ-abstraction and not is the logical negation, often written in linear logic using the superscript $\perp$). The resulting formula,

```
pi X\(pi M\(pi N\(not (rev N (X::M)) :- not (rev (X::N) M)))) =>
not (rev k nil) -: not (rev nil l),
```

is thus provable from $\mathcal{P}$. This formula is logically equivalent to the following formula (linear implications and their contrapositives are equivalent in linear logic).

$\sim A$ is $A^{\perp}$

$(p^{\perp} \multimap q^{\perp}) \multimap\!\circ (q \multimap p)$

```
pi X\(pi M\(pi N\(rev (X::N) M :- rev N (X::M)))) =>
rev nil l -: rev k nil
```

Since this code is provable and since the constant rev is not free in $\mathcal{P}$, we can universally generalize over it; that is, the following formula is also provable.

```
pi rev\(
    pi X\(pi M\(pi N\(rev (X::N) M :- rev N (X::M)))) =>
     rev nil l -: rev k nil)
```

From this goal and the definition of reverse (and α-conversion) we can prove (reverse k l). Hence, reverse is symmetric. ∎

This proof should be considered elementary since it involves only simple linear logic identities and facts. Notice that there is no direct use of induction. The two symmetries mentioned above in the informal proof are captured in the higher-order substitution x\y\(not (rev y x)): the switching of the order of bound variables captures the vertical flip and linear logic negation (via contrapositives) captures the the horizontal flip.

3 Meta-programming and meta-logic

An exciting area of specification is that of specifying the meaning and behavior of programs and programming languages. In such cases, the code of a programming language must be represented and manipulated, and it is valuable to introduce the terms *meta-language* to denote the specification language and *object-language* to denote the language being specified.

Given the existence of two languages, it is natural to investigate the relationship that they may have to one another. That is, how can the meaning of object-language expressions be related to the meaning of meta-level expressions. One of the major accomplishments in mathematical logic in the first part of this century was achieved by K. Gödel by probing this kind of reflection, in this case, encoding meta-level formulas and proofs at the the object-level [7].

Although much of the work on meta-level programming in logic programming has also been focused on reflection, this focus is rather narrow and limiting: there are many other ways to judge the success of a meta-programming language apart from its ability to handle reflection. While a given meta-programming language might not be successful at providing novel encodings of itself, it might provide valuable and flexible encodings of other programming languages. For example, the π-calculus provides a revealing encoding of evaluation in the λ-calculus [17], evaluation in object-oriented programming [28], and interpretation of Prolog programs [12]. Even the semantic theory of the π-calculus can be fruitfully exploited to probe the semantics of encoded object-languages [27]. While it has been useful as a meta-language, it does not seem that the π-calculus would yield an interesting encoding of itself.

Similarly, λProlog has been successful in providing powerful and flexible specifications of functional programming languages [8, 21] and natural deduction proof systems [5]. Forum has similarly been used to specify sequent calculi and various features of programming languages [2, 14]. It is not clear, however, that λProlog or Forum would be particularly good for representing their own operational semantics.

> *Observation 2.* A meta-programming language does not need to capture its own semantics to be useful. More importantly, it should be able to capture the semantics of a large variety of languages and the resulting encoding should be direct enough that the semantics of the meta-language can provide semantically meaningful information about the encoded object-language.

A particularly important aspect of meta-programming is the choice of encodings for object-level expressions. Gödel used natural numbers and the prime factorization theorem to encode syntactic values: an encoding that does not yield a transparent nor declarative approach to object-level syntax. Because variables in logic programming range over expressions, representing object-level syntax can be a particularly simple, at least for certain expressions of the object language. For example, the meaning of a type in logic programming, particularly types as they are used in λProlog, is a set of *expressions* of a given type. In contrast, types in functional programming (say, in SML) generally denote sets of *values*. While the distinction between expressions and values can be cumbersome at times in logic programming (`2 + 3` is different than `5`), it can be useful in meta-programming. This is particularly true when dealing with expressions of functional type. For example, the type `int -> int` in functional programming denotes functions from integers to integers: checking equality between two such functions is not possible, in general. In logic programming, particularly in λProlog, this same type contains the code of expressions (not functions) of that type: thus it is possible to represent the syntax of higher-order operations in the meta-programming language and meaningfully compare and compute on these codes. More generally, meta-level types are most naturally used to represent object-level syntactic categories. When using such an encoding of object-level languages, meta-level unification and meta-level variables can be used naturally to probe the structure of object-level syntax.

> *Observation 3.* Since types and variables in logic programming range over expressions, the problem of naming object-level expressions is often easy to achieve and the resulting specifications are natural and declarative.

4 Higher-order abstract syntax

In the last observation, we used the phrase "often easy to achieve." In fact, if object-level expressions contain bound variables, it is a common observation that

representing such variables using only first-order expressions is problematic since notions of bound variable names, equality up to α-conversion, substitution, *etc.*, are not addressed naturally by the structure of first-order terms. From a logic programming point-of-view this is particularly embarrassing since all of these notions are part of the meta-theory of quantification logic: since these issues exist in logic generally, it seems natural to expect a logical treatment of them for object-languages that are encoded into logic. Fortunately, the notion of *higher-order abstract syntax* is capable of declaratively dealing with these aspects of object-level syntax.

Higher-order abstract syntax involves two concepts. First, λ-terms and their equational theory should be used uniformly to represent syntax containing bound variables. Already in [3], Church was doing this to encode the universal and existential quantifiers and the definite description operator. Following this approach, instantiation of quantifiers, for example, can be specified using β-reduction.

The second concept behind higher-order abstract syntax is that operations for composing and decomposing syntax must respect at least α-conversion of terms. This appears to have first been done by Huet and Lang in [11]: they discussed the advantages of representing object-level syntax using simply typed λ-terms and manipulating such terms using matching modulo the equational rules for λ-conversion. Their approach, however, was rather weak since it only used matching (not unification more generally). That restrictions made it impossible to express all but the simplest operations on syntax. Their approach was extended by Miller and Nadathur in [15] by moving to a logic programming setting that contained $\beta\eta$-unification of simply typed λ-terms. In that paper the central ideas and advantages behind higher-order abstract syntax are discussed. In the context of theorem proving, Paulson also independently proposed similar ideas [20].

In [23] Pfenning and Elliot extended the observations in [15] by producing examples where the meta-language that incorporated λ-abstractions contained not just simple types but also product types. In that paper they coined the expression "higher-order abstract syntax." At about this time, Harper, Honsell, and Plotkin in [9] proposed representing logics in a dependent typed λ-calculus. While they did not deal with the computational treatment of syntax directly, that treatment was addressed later by considering the unification of dependent typed λ-expressions by Elliott [4] and Pym [25].

The treatment of higher-order abstract syntax in the above mentioned papers had a couple of unfortunate aspects. First, those treatments involved unification with respect to the full $\beta\eta$-theory of the λ-calculus, and this general theory is computational expensive. In [11], only second-order matching was used, an operation that is NP-complete; later papers used full, undecidable unification. Second, various different type systems were used with higher-order abstract syntax, namely simple types, product types, and dependent types. However, if abstract syntax is essentially about a treatment of bound variables in syntax, it should have a presentation that is independent from typing.

The introduction of L_λ in [13] provided solutions to both of these problems.

First, L_λ provides a setting where the unification of λ-terms is decidable and has most general unifiers: it was shown by Qian [26] that L_λ-unification can be done in linear time and space (as with first-order unification). Nipkow showed that the exponential unification algorithm presented in [13] can be effectively used within theorem provers [19]. Second, it was also shown in [13] that L_λ-unification can be described for *untyped* λ-terms: that is, typing may impose additional constraints on unification but L_λ-unification can be defined without types. Thus, it is possible then to define L_λ-like unification for various typed calculi [22].

> *Observation 4.* L_λ appears to be one of the weakest settings in which higher-order abstract syntax can be supported. The main features of L_λ can be merged with various logical systems (say, λProlog and Forum), with various type systems (say, simple types and dependent types) [21], and with equational reasoning systems [18, 24].

While existing implementations of λProlog, Isabelle, Elf, and NuPRL all make use of results about L_λ, there is currently no direct implementation of L_λ. It should be a small and flexible meta-logic specification language.

References

[1] J.M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

[2] Jawahar Chirimar. *Proof Theoretic Approach to Specification Languages.* PhD thesis, University of Pennsylvania, February 1995.

[3] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[4] Conal Elliott. Higher-order unification with dependent types. In *Rewriting Techniques and Applications*, volume 355, pages 121–136. Springer-Verlag LNCS, April 1989.

[5] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

[6] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[7] Kurt Gödel. On formally undecidable propositions of the principia mathematica and related systems. I. In *Martin Davis, The Undecidable.* Raven Press, 1965.

[8] John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, April 1993.

[9] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY, June 1987.

[10] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[11] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[12] Benjamin Li. A π-calculus specification of Prolog. In *Proc. ESOP 1994*, 1994.

[13] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991. λ-Prolog.

[14] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual Symposium on Logic in Computer Science*, pages 272–281, Paris, July 1994.

[15] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.

[16] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[17] Robin Milner. Functions as processes. 17th Int. Coll. Automata, Languages and Programming Warwick University, UK, LNCS 443, pp. 167–180, Springer Verlag July 1990.

[18] Tobias Nipkow. Higher-order critical pairs. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 342–349. IEEE, July 1991.

[19] Tobias Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 64–74. IEEE, June 1993.

[20] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[21] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–321, Monterey, CA, June 1989.

$\forall \lambda x((p\,x \supset q\,x) \wedge r) = \forall \lambda y(R \wedge S)$
no unifier modulo α-conversion.

[22] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 74–85. IEEE, July 1991.

[23] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[24] Christian Prehofer. *Solving Higher-Order Equations: From Logic to Programming.* PhD thesis, Technische Universität München, 1995.

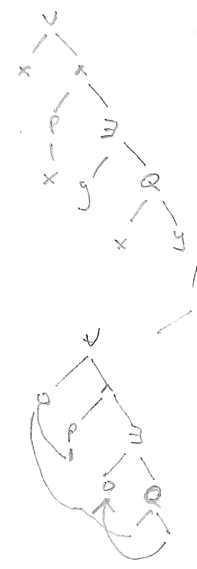
[25] David Pym. *Proofs, Search and Computation in General Logic.* PhD thesis, LFCS, University of Edinburgh, 1990.

[26] Zhenyu Qian. Linear unification of higher-order patterns. In J.-P. Jouannaud, editor, *Proc. 1993 Coll. Trees in Algebra and Programming.* Springer Verlag LNCS, 1993.

[27] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, May 1994.

[28] David Walker. π-calculus semantics of object-oriented programming languages. LFCS Report Series ECS-LFCS-90-122, University of Edinburgh, October 1990.

Church style ⟹ $\forall x\,(Px \wedge \exists y\, Qxy)$



Parse trees

use first-order terms and structural induction

Abstract syntax

What here?
unification modulo α conversion ||

de Bruijn numerals are only a good implementation.

$= \forall \lambda y\,(Ry \wedge S)$
$R \to \lambda w\,(p\,w \supset q\,w)$
$S \Rightarrow r$ this unifier.

unification using β and η unification is unsolvable. Using only β₀ conversion these ~~[illegible]~~ unification is solvable. unification is modulo β₀ conversion.

$(\beta_0)\ (\lambda x B)x \to B$
$(\beta_0)\ (\lambda x B)y \to B[y/x]$

PARALLEL LOGIC PROGRAMMING WITH EXTENSIONS

β₀ expansion is a strategy

Luís Moniz Pereira and José C. Cunha
Project Coordinators
Departamento de Informática
Universidade Nova de Lisboa
P-2825 Monte da Caparica, Portugal
Luís Damas
Project Coordinator
Laboratório de Inteligência Artificial e
Ciência dos Computadores
Universidade do Porto
R. Campo Alegre 823, 4100 Porto, Portugal

Abstract

A programming language is a tool and a vehicle for applications. Logic Programming has proven to be a very fruitful paradigm. Recognizing the need to promote the evolution of Prolog towards a more expressive new logic programming language, a large medium-term national research project was recently started under the authors' coordination involving a large body of investigators from their home institutions. Extensions to Logic Programming are being developed with new forms of computational reasoning, with explicit negation, constraint programming, and parallelism and distribution support. The resulting language, PROLOPPE, will integrate the above aspects in the form of a trully efficient implementation that exploits innovative techniques, including joint implicit and explicit parallelism, and distribution over heterogenous multiple processor architectures. This language will be used in a wide variety of applications such as desision support systems, natural language processing, diagnosis, scheduling, and robot cooperation. In this paper we overview the main topics behind the research in the PROLOPPE project.

This paper is an edited version, by the authors, of part of the original proposal of the PROLOPPE Project which included contributions from C. Damásio, F. Menezes, J. C. Cunha, J. J. Alferes, L. M. Pereira, P. Barahona, P. D. Medeiros, S.P. Abreu, from the Universidade Nova de Lisboa, and A. Matos, A. P. Tomás, F.

Silva, J. P. Leal, L. Damas, L. Lopes, M. Filgueiras, N. Moreira, R. Reis and V. S. Costa from the Universidade do Porto, and Miguel Calejo from Servisoft.

1 Introduction

Prolog, albeit its great success, has a number of limitations both at the language and at the execution levels. We propose to overcome some of these limitations by relying and improving on recent results in the semantics of logic programming languages, where we have ourselves made relevant contributions. We think it is now time for a new step in this direction, with the proposal of a language augmenting the expressive and computational power of Prolog. Our main objective is to promote the evolution of Prolog towards a more expressive new logic programming language (Proloppe), tied to an efficient execution environment that exploits innovative implementation techniques, including joint implicit and explicit parallelism, and distribution over heterogenous multiple processor architectures. Extensions to Logic Programming will be developed with new forms of computational reasoning, with explicit negation, constraint programming, and parallelism and distribution support.

Prolog supports a subset of Logic only, and in particular it does not include explicit negation in the facts and in the rules. Increased expressive power of the language, and a wider scope for its applications will be possible by including explicit negation as well as default negation, and exploiting new forms of non-monotonic reasoning (a promising field with increasing expansion at the international scale). Handling contradictions and automated methods for belief revision also brings in increased functionalities. Implementation of these functionalities requires the support of construtive negation, integrity constraints, and disjunction. Constraint-based programming methods for solving linear Diophantine equations have been developed by researchers in our groups, and are proved to be among the most efficient methods known. These and other methods, namely incremental hierarchical constraint solvers for finite domains and the naturals, will be integrated in the implementation and will support rapid prototyping of efficient applications in multiple domains (e.g. scheduling and time-tabling). Currently, these applications demand highly specialized programs which are hard to specify and mantain. Integrating forms of non-monotonic reasoning and numerical constraints in a logic language opens possibilities for innovation in the area of Decision Support Systems.

A first efficient implementation including such extensions will be developed, following our past experience, and aiming at overcoming several limitations of current logic programming systems, regarding problem solving in non-trivial AI applications, and in distributed AI. The project encompasses the development of new techniques concerning the following aspects: optimized Prolog compilation, execution models for the proposed extensions, joint exploitation of implicit and explicit parallelism over heterogeneous multiple processor architectures, and tools for program development with sequential and parallel execution, with support for performance measurement, debugging and visualization.

We address a diversity of complex application domains requiring a programming system with great expressiveness, declarativity, efficiency, and intelligence in the execution strategies. These aspects will contribute to the evaluation of the developed tools, as well as to strenghten the logic programming abilities in order to address non-trivial problem solving. Application development will also be pursued in other initiatives, extending its impact at the national and international scopes.

In the following sections we summarize the fundamental issues that are being addressed in this project.

2 Semantics for negation

Recently, several authors have underscored the importance of extending logic programming (LP) with a second kind of negation $\neg$, for use in knowledge representation, deductive databases and nonmonotonic reasoning (NMR) [GL90, GL92, Ino91, Kow90, KS90, PW90, PAA91b, PAA91d, PAA92b, PDA93b, PDA93c, PDA93a, PAA93, Wag91]. [BG93] makes an overview of the use of such programs in knowledge representation and NMR. Different semantics for extended logic programs with $\neg$–negation (ELP) have appeared [DR91, GL90, KS90, PA92, PAA91a, PAA92a, Prz90, Prz91a, Sak92, Wag91]. Each of these semantics is a generalization for ELP of either the stable models semantics (SM) [GL88], or the well–founded semantics (WFS) [GRS91] of normal programs.

In [Prz90, Dix91, Dix92] SM and WFS are contrasted, and it is argued that, by its structural properties, WFS is more suitable for an implementation (unlike SM, it is possible to define for WFS both bottom–up and top–down procedures [PAA91c, CW92, Ros92, BD93]). To deal with the problem of floundering, the top–down procedures need to treat negation as failure goals by means of procrastination, common to that of constraints and deterministic priority, and/or constructive methods [TS86, Prz89, KT88, CW92, Ros92, BD93, Bol93].

[AP92] contrasts some of these semantics regarding their use of $\neg$–negation, where distinct meanings of $\neg$ are identified (explicit, strong and classical), and argues that, by its properties, explicit negation is preferable. By being a generalization of WFS for ELP which uses explicit negation, WFSX [PA92] appears as a natural candidate for the base semantics of our ELP PROLOPPE implementation. [Alf93] make an extensive study on WFSX, and its relationship with several NMR formalisms. In [AP93b] WFSX is defined in terms of a "logic of belief and provability', and compared with the works of [Che93, MT93, LS93]. This open the way for contradiction removal (cf. below), and for generalizing WFSX to disjunctive programs, in the spirit of [BLM90, Prz91c, Prz91b, Prz93], where several semantics for disjunctive normal programs are defined.

By generalizing LP with either explicit negation or integrity constrains, a new issue arises: how to deal with contradiction. [DR91, Jon91, PA93b, PAA91a, PAA92a] present several proposal for that issue. [AP93c, PA93a] distinguish between two generic approaches to deal with contradiction: one consists in *avoiding* it; the other in *removing* it. The definition on procedures for removing contradiction has been generalized to deal with two valued revisions [PDA93b], and to deal with preference among revisions [DNP93], with application to diagnosis, updates, and debugging [PDA93c].

Despite all the above mentioned effort on the theoretical study on ELP semantic and its application domains, to date there is no efficient implementation of these semantics, nor even a formal specification of their procedures. The ELP implementation of this proposal is intended to fill in this gap, and to allow for a practical application of ELP for problems of the domains studied.

3 Constraint Logic Programming

The characteristics of LP, namely its declarative nature, makes it particularly suitable to the specification of a large number of constraint satisfaction problems. Nevertheless, the resolution principle, the basis of LP, is insufficient to handle efficiently these problems, since it does not take advantage from the specificity of some domains (namely numeric) nor from the characteristics of the operations defined on these domains. Several extensions have thus been proposed to LP in the last few years that, without jeopardizing its declarative nature, allow a much better performance in solving these problems. In general, these languages extend LP to Constraint Logic Programming (CLP), by replacing the resolution principle by more powerful constraint solving method in some specialised domain.

Solving linear constraints on finite domains may also be done by exploring the equivalence to the problem of solving systems of linear equations over the natural numbers (Diophantine equations) and using the specific methods developed for it. Most of the recent research work on Diophantine equations is related with the development of algorithms for unification of terms with associative and commutative functors (AC-unification) and with the field of Term Rewriting Systems [Dom91]. The use in the implementation of a CLP system of one of the methods for solving a system of Diophantine equations is under research [Con93]. Other recent results, for a single equation, are described in [TFar] and [FT93] and correspond to the fastest methods known to date.

The topic of constraints over algebras of rational trees extends term unification, in a decidable way [Mah88, CL89], to the resolution of first order formulas with equality as unique predicate symbol. Extensions to Prolog in this line, are Prolog II [Col82], Prolog III, and more recent, systems as $CLP(\mathcal{FT})$ [Smi91], where universally quantified disequalities are used to allow logic programs with constructive negation. On the other hand, as was pointed out in [DMV93], the standard algebra of rational trees has a close relationship with the standard model for features

logics, [Smo89], which were establish in order to formalize feature based grammar formalisms that have emerged in the Computational Linguistics community over the past few years. From a practical point of view, the fact that the satisfiability problem (in these domains) is NP-hard tends to manifest itself in a dramatic way in practical applications, motivated several specialized algorithms to minimize this problem [Kas87, ED88, MK91].

In [DV92] it was argued that any practical approach to the satisfiability problem should use factorization techniques to reduce the size of the input formulae to which any complete algorithm for satisfiability is applied, since such factorization can reduce by an exponential factor the overall computational cost of the process. In [DMV93, DMB93] were described more factorization techniques and a complete rewrite system for satisfiability was provided.

4 Compilers for Prolog with Extensions

Prolog adapts well to conventional computer architectures. Prolog's selection function and search rule are simple operations. Moreover, the fact that Prolog only uses terms means that the state of the computation can be coded quite efficiently. The basis for most of the current implementations of logic programming languages is the Warren Abstract Machine [War83], or *WAM*, an "abstract machine" useful as a target for the compilation of Prolog because it can be implemented very efficiently in most conventional architectures. Recent efforts on the implementation of Prolog have tried to improve further the performance by using direct compilation to native code and global analysis [Van90, Tay90]. Native code systems gain performance by by-passing the emulator. They can also perform machine-level optimisations. Global analysis provides information on how arguments are actually used during execution. Its most common uses are in the further specialisation of unification and in more sophisticated indexing.

The above-mentioned systems are not portable, that is, they usually depend on knowledge of some computer architectures. A portable approach, direct generation of "C" code, has been proposed by Debray among others, but can lead to the generation of huge and hard to compile "C" programs. We believe that such work could benefit from well-established work on portable C compilers.

4.1 New execution models

From the very beginnings of logic programming there has been a desire to obtain good execution models [Kow79] [PP79] [Col86] [Nai85]. More recently the need for such models has been made even clearer by the new goal of exploring parallelism in logic programs. One such important model is the Warren's Basic Andorra Model [War88], used in the the Andorra-I system [SCWY91b, SCWY91c]. In this model determinate goals (that is, goals for which at most a single clause can match) can be selected first and run in and-parallel. When no such goals are available, the system can try the several alternatives to a non-determinate goal in or-parallelism. Besides the parallelism, the selection functions most natural to the Basic Andorra Model have a very useful form of implicit coroutining [SC93], which has been exploited in several Andorra-I applications [Yan89, GY92] and in the Pandora language [Bah93]. Note that Andorra-I can only exploit and-parallelism between determinate goals. Warren's Extended Andorra Model (EAM) [War89] lifts this restriction and allows a general form of and-parallelism. The EAM gives a set of general rewrite rules for logic programs, which can be subject to different control schemes. The EAM was a basis for the Kernel Andorra Prolog (KAP) [HJ90] framework which is instantiated in the AKL language, proposed by Janson and Haridi [JH91]. In these languages, guards (such as commit guards, cut guards and wait guards) are used to control computation, which may be nondeterministic. Both or-parallelism, and and-parallelism between non-determinate (and determinate) goals can be exploited. Moreover, the search space can be much reduced over traditional Prolog systems.

Further improvements to AKL's search rule have been performed by Abreu, Pereira and Codognet [APC92a]. The authors have studied failure-driven configuration reordering, which can be seen as an application of the first-fail principle to the unfolding of an AKL computation. This shows that And-Or tree Rewriting systems (AORS), which encompasses both AKL and the EAM, provide a fertile base for the exploitation of a-posteriori search-space pruning, i.e. pruning part of the search-space as a consequence of the execution of another portion of the program. This approach complements the a-priori search-space pruning that comes as a result of constraint propagation, another mechanism present in AKL.

The differences between Prolog and Andorra-I are more striking. Andorra-I does in fact inherit most of its implementation techniques from Parlog [Cra88] and KL1 [SSM$^+$87]. Andorra-I's abstract machine and compiler are described by Santos Costa [SC93] (note that in practice much of the difficulties to be addressed in Andorra-I are due to parallelism support, handled by Yang's engine and by the several schedulers[BRSW91, Dut91]). Andorra-I incorporates some optimisations, the ones considered particularly important for Andorra-I's main goal to run real applications. The compiled Andorra-I is not as optimised as current Prolog systems, being a more complex and a newer system. Great improvements can be obtained by using the new techiques that are being developed for Prolog, plus the new techiques developed for the committed-choice languages [TB93].

The implementation of AKL and of the EAM also brings some new problems. Janson and Montelius have a prototype implementation [JM92], but again several optimisations will be needed for one such system to compete with current Prolog systems. Note that AKL (and EAM) can be described in terms of and-boxes, and or-boxes (several types may exist). These boxes are expanded during forward execution, but their configuration must be reorded upon failure. This can be made more effective through the guidance of the reordering scheme by a binding-dependency

maintenance system; such is the object of the AKL/IP (for AKL with Intelligent Pruning) system outlined in [AP93a] and [APC92b], currently being worked on in Lisbon. AKL/IP is currently being implemented using Janson and Montelius' prototype [JM92] as a basis, being thus a sequential implementation. It can be argued that a computational model based on rewrite rules for and-or trees (as is the case with AORS's) is more suited to dependency-directed search-space pruning than systems using a Prolog-like selection rule, because the former provides a built-in[1] mechanism to describe suspension of goals and can cope more gracefully with changes to the relative ordering of goals at run-time.

5 Implicit and Explicit Parallelism

Parallel logic programming systems obtain high-performance by exploiting different forms of parallelism. Both implicit and explicit parallelism are available in logic programming languages. In explicit systems such as Delta Prolog [PMCA86] special types of goals (events and splits in Delta Prolog) are available to control parallelism. Implicit parallelism can be obtained through the parallel execution of several resolvents arising from the same query, *or-parallelism*, or through the parallel resolution of several goals, *and-parallelism*. All these forms of parallelism can be explored according to very different strategies. We next discuss the most important techniques now available to exploit parallelism in logic programs.

5.1 Implicit Parallelism

Both or-parallelism and and-parallelism have been exploited successfully in logic programming systems. Whereas or-parallel systems exploit much the same parallelism, and differ mainly in the way they represent the search space, quite a few different forms of and-parallelism have been recognised. Arguably, some of the most important approaches are the following. Systems implementing the committed-choice languages, exploit parallelism between goals that have commited to a single clause. Independent and-parallelism systems, such as &-Prolog [HG90], only run independent goals, whose computation should not interfere, in parallel. Andorra-I [SCWY91a] exploits and-parallelism between goals that are *determinate*. The latter idea has been generalised in the Extended Model, and in the AKL language[JH91].

5.2 Parallel Implementation of the EAM and AKL

Andorra-I can only exploit and-parallelism between determinate goals. Both the EAm and AKL lift this restriction. AKL and the EAM thus share the property that the computation may be carried out in parallel more naturally than in other parallel

[1]i.e. intrinsic to the execution model, not as an add-on such as can be found in some Prolog systems, or in multi-sequential parallel implementations.

logic programming systems: the approach of having the run-time data structures organized as a tree of and-boxes and choice-boxes, the requirements on quietness of pruning guards, together with the fact that and-boxes have their own store lead to potentially better locality properties, making such a system suitable for both coarse and fine-grained parallelism. Indeed, the gains on a parallel implementation of AKL are very attractive, and Moolenaar[VAMD91] has recently implemented a parallel prototype for of AKL.

Finally, the promising experimental results obtained with the preliminary prototype of the AKL/IP system prompt us to look into a true parallel implementation.

5.3 Implicit Parallelism for Distributed Memory Systems

Recently, new parallel architectures, namely distributed shared memory architectures, have been proposed and built (e.g. the KSR and EDS parallel machines). Although, in these architectures, the memory is physically distributed there is software and hardware support for a shared virtual memory computation model. These architectures combine the advantages of large number of processors (scalability) with the advantage of shared memory, and are therefore ideal targets for the parallel execution of Prolog programs. These architectures are quite recent and few parallel logic programming systems have been developed that understand the new issues. One of the first models has been designed and implemented by Silva [Sil93], and shown successful execution for or-parallelism in one of these architectures.

5.4 Explicit Parallelism

This approach consists in the definition of constructs for the explicit specification of sequenciality and concurrency, synchronization and non-determinism in a logic programming language.

The diversity of proposals that have arisen in the past ten years have two principal aims: (i) search for increased flexibility in the specification of parallelism, versus implicit parallelism as supported by a compiler and/or run-time system; (ii) the need for suitable constructs for the specification of distributed systems. A large number of problems are naturally modelled by multiple concurrent interacting processes where the data structures and/or the entities that solve the problem are spacially distributed.

6 Programming environment

Mostly since the 80s several efforts have been made to build powerful logic programming environments, in order to meet the expectations brought by the innovative Prolog language in the 70s [DCLY93]. One of the areas where more promising results were found, by capitalizing on logic programming's own caracteristics, was declarative debugging, which has grown into an area with autonomous scientific workshops, such as the recent [FN93], [Cal92]. So far one of the greatest eforts towards an environment integrating innovative tools has been the ESPRIT ALPES project [ANPR89], which went on between 1986 and 1988. From that work and from its sequel by the UNL team several prototype Prolog environments were developped (X-Prolog, MacLogic, StepOnProlog), associated to various Prolog implementations (Apple Computer's Logic Manager, UNL's own nanoProlog, Universidade do Porto's YAP, University of Edinburgh's C-Prolog), sometimes in the context of external R&D contracts (Apple, ENIDATA, Digital, Softlog/NeXT, among others).

In this project we are further improving the development tools in the environment, including low-level analyzers for sequential and parallel execution, declarative debugging, browsers, graphics tools and interface buiders.

7 Tasks Overview

This project represents a significant effort, at national scale, towards promoting the Logic Programming paradigm in several directions: theory, language, execution models, implementations and applications. It spans a large body of researchers, and it will stimulate a diversity of teaching and training activities.

Besides the semantic definitions for the new PROLOPPE language, we will produce the first usable implementation of the language and a development environment to be made available to the international academic community, promoting the exploitation of AI applications. This implementation will include and extend the results that the proponents have been achieving, regarding a better Prolog execution model, applied to the support of more advanced semantics and a joint exploitation of implicit and explicit parallelism. The tasks below will produce specifications of language extensions, models and prototypes, and the prototypes themselves, which will be demonstrated and made available during the project.

- Explicit Negation and Logic Programming with Non-Monotonic Reasonning, including contradiction removal, with constructive negation and disjunction.
- Constraint Logic Programming: constraint resolution methods are investigated and used in the implementation of constraint logic programming languages: (a) resolution methods for the linear Diophantines equation systems and other more general constraints concerning naturals and finite domains; (b) incremental hierarchical constraint solvers over natural and finite domains.
- Execution Models: optimized compilation of the Prolog model will be explored, since our past experience in the implementation of conventional Prolog shows that great improvements can still be achieved when implementing the language. This includes: (a) the design of an Intermediate Computer Description, oriented to the underneath architecture; (b) the implementation of

a compiler with procedure and intra-procedure level optimization, featuring unfolding, choice points elimination and mode or sequentiality detection; (c) extensibility support to provide the proposed extensions to the logic language.

- We will develop other execution models and search strategies in the following aspects: (a) optimization of the search process based in "Intelligent Pruning", a method that resembles Prolog's intelligent backtracking, but applied to AKL (AKL/IP) execution model; (b) sequential and parallel implementation of the AKL/IP language; (c) application of the AKL/IP execution model to non-monotonic reasoning, as support to an implementation support of the previous extensions.
- Implicit and Explicit Parallelism: implicit parallelism of the OR and AND types will be explored over shared memory and distributed memory architectures, integrated with other forms of parallelism suitable to the support of distributed logic programming, its application in Distributed AI, and its implementation over heterogenous multiprocessors.
- Development Environment: we will integrate the acomplished extensions in an environment with a set of user support tools, such as: (a) low-level analyser with performance measuring tools, and sequential and parallel execution tracing; (b) declarative debugger; (c) browser; (d) graphic library and specification languages for system interaction; (e) visualization of distributed computations.
- Some applications will be developed to evaluate, test, promote PROLOPPE: (a) syntactic analyser for natural language (for the testing of system use in new formalisms based on constraints like the HPSG); (b) Constraints and Time-Tabling; (c) To diagnosis of distributed artificial intelligence and non-monotonic reasoning.

  Other aplications of non-monotonic reasoning are forseen, e.g. to planning and to natural language, not carried out within the project, but evaluated by institutional colleges of team members. The results of this evalutation will be report.

Acknowledgments. This research is nationally supported in part by Programa CIENCIA and project PROLOPPE of the PRAXIS XXI programme.

References

[Alf93] José Júlio Alferes. *Semantics of Logic Programs with Explicit Negation.* PhD thesis, Universidade Nova de Lisboa, October 1993.

[ANPR89] J.A.S. Alegria, A. Natali, N. Preston, and C. (editors) Ruggieri. Alpes final report (esprit p973 project). Technical report, Universidadade Nova de Lisboa, 1989.

[AP92] J. J. Alferes and L. M. Pereira. On logic program semantics with two kinds of negation. In K. Apt, editor, *Int. Joint Conf. and Symp. on LP*, pages 574–588. MIT Press, 1992.

[AP93a] Salvador Abreu and Luís Moniz Pereira. Design for akl with intelligent pruning. In Roy Dyckhoff, editor, *Workshop on Extensions of Logic Programming*, pages 1–6. University of St. Andrews, Scotland, 1993.

[AP93b] J. J. Alferes and L. M. Pereira. Belief, provability, and logic programs (draft). Technical report, CENTRIA, 1993. Submitted to KR'94.

[AP93c] J. J. Alferes and L. M. Pereira. Contradiction: when avoidance equal removal. Part I. In R. Dyckhoff, editor, *4th Int. Ws. on Extensions of LP*, pages 7–16. Univ. of St. Andrews, 1993.

[APC92a] Salvador Abreu, Luís Moniz Pereira, and Philippe Codognet. Improving backward execution in the andorra family of languages. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 384–398, Washington, USA, 1992. The MIT Press.

[APC92b] Salvador Abreu, Luís Moniz Pereira, and Philippe Codognet. Improving Backward Execution in Non-deterministic Concurrent Logic Languages. In Ryuzo Hasegawa and Mark Stickel, editors, *Workshop on Automated Deduction, Fifth Generation Computer Systems.* Institute for New Generation Computing, 1992.

[Bah93] Reem Bahgat. *Non-Deterministic Concurrent Logic Programming in Pandora.* World Scientific, 1993.

[BD93] R.N. Bol and L. Degerstedt. Tabulated resolution for well-founded semantics. In *Proc. Int. Logic Programming Symposium'93.* MIT Press, 1993.

[BG93] C. Baral and M. Gelfond. Logic programming and knowledge representation (draft). Technical report, University of Texas at El Paso, 1993.

[BLM90] C. Baral, J. Lobo, and J. Minker. Generalized disjunctive well-founded semantics. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *10th Int. Conf. on Automated Deduction*, pages 102–116. LNAI 449, Spriger–Verlag, 1990.

[Bol93] R.N. Bol. Loop checking and negation. *Journal of Logic Programming*, 15(2):147–175, 1993.

[BRSW91] Anthony Beaumont, S Muthu Raman, Péter Szeredi, and David H. D. Warren. Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991.

[Cal92] Miguel Calejo. *A Framework for Declarative Prolog Debugging.* PhD thesis, Universidade Nova de Lisboa, March 1992.

[Che93] J. Chen. Minimal knowledge + negation as failure = only knowing (sometimes). In L. M. Pereira and A. Nerode, editors, *2nd Int. Ws. on LP & NMR*, pages 132–150. MIT Press, 1993.

[CL89] H. Comon and Pierre Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:317–425, 1989.

[Col82] Alain Colmerauer. Prolog and infinite trees. In S. A. Tarnlund, editor, *Logic Programming.* AC, 1982.

[Col86] Alain Colmerauer. Theoretical Model of Prolog II. In Michel van Caneghen and David H. D. Warren, editors, *Logic Programming and its Applications*, pages 3–31. Ablex Publishing Corporation, 1986.

[Con93] Evelyne Contejean. Solving linear diophantine constraints incrementally. In D. S. Warren, editor, *Logic Programming: Proceedings of the 10th International Conference on Logic Programming*. MIT Press, 1993.

[Cra88] J. A. Crammond. *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors*. PhD thesis, Heriot-Watt University, Edinburgh, May 1988. Research Report PAR 88/4, Dept. of Computing, Imperial College, London.

[CW92] W. Chen and D. S. Warren. A goal–oriented approach to computing well–founded semantics. In K. Apt, editor, *Int. Joint Conf. and Symp. on LP*, pages 589–603. MIT Press, 1992.

[DCLY93] M. Ducassé, B. Charlier, Y. Lin, and Ü. Yalcinalp, editors. *Post Conf. Ws. on Logic Programming Environments*. ILPS'93, 1993.

[Dix91] J. Dix. Classifying semantics of logic programs. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *LP & NMR*, pages 166–180. MIT Press, 1991.

[Dix92] J. Dix. A framework for representing and characterizing semantics of logic programs. In B. Nebel, C. Rich, and W. Swartout, editors, *3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, 1992.

[DMB93] Luís Damas, Nelma Moreira, and Sabine Broda. Resolution of constraints in algebras of rational trees. In Miguel Filgueiras and Luís Damas, editors, *Progress in Artificial Intelligence — 6th Portuguese Conference on Artificial Intelligence*, volume 727 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1993.

[DMV93] Luís Damas, Nelma Moreira, and Giovanni B. Varile. The formal and computational theory of complex constraint solution. In C. Rupp, M. Rosner, and R. Johnson, editors, *Constraints, Language, and Computation*. Academic Press, London, 1993.

[DNP93] C. V. Damásio, W. Nejdl, and L. M. Pereira. REVISE: An extended logic programming system for revising knowledge bases (draft). Technical report, CENTRIA and RTHW–Aachen, 1993. Submitted to KR'94.

[Dom91] Eric Domenjoud. *Outils pour la Déduction Automatique dans les Théories Associatives-Commutatives*. PhD thesis, Université de Nancy I, 1991.

[DR91] P. M. Dung and P. Ruamviboonsuk. Well founded reasoning with classical negation. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *LP & NMR*, pages 120–132. MIT Press, 1991.

[Dut91] Inês Dutra. A Flexible Scheduler for the Andorra-I System. In *LNCS 569, ICLP'91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, pages 70–82. Springer-Verlag, June 1991.

[DV92] Luís Damas and Giovanni B. Varile. On the satisfiability of complex constraints. In *Coling'92*, Nantes, France, 1992.

[ED88] A. Eisele and J. Dörre. Unification of disjunctive feature descriptions. In *26th Annual Meeting of the Association for Computational Linguistics*, Buffalo, New York, 1988.

[FN93] P. Fritzson and H. Nilsson, editors. *1st Int. Ws. on Automatic Algorithmic Debugging, AADEBUG'93*. Preproceedings by Linkoping Univ., 1993. To appear in Springer–Verlag LNCS.

[FT93] Miguel Filgueiras and Ana Paula Tomás. Fast methods for solving linear Diophantine equations. In M. Filgueiras and L. Damas, editors, *Proceedings of the 6th Portuguese Conference on Artificial Intelligence*, pages 297–306. LNAI 727, Springer–Verlag, 1993.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *5th Int. Conf. on LP*, pages 1070–1080. MIT Press, 1988.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In Warren and Szeredi, editors, *7th Int. Conf. on LP*, pages 579–597. MIT Press, 1990.

[GL92] M. Gelfond and V. Lifschitz. Representing actions in extended logic programs. In K. Apt, editor, *Int. Joint Conf. and Symp. on LP*, pages 559–573. MIT Press, 1992.

[GRS91] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[GY92] Steve Gregory and Rong Yang. Parallel Constraint Solving in Andorra-I. In *International Conference on Fifth Generation Computer Systems 1992*, pages 843–850. ICOT, Tokyo, Japan, June 1992.

[HG90] M. V. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[HJ90] Seif Haridi and Sverker Jansson. Kernel Andorra Prolog and its Computational Model. In D.H.D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 31–46. MIT Press, 1990.

[Ino91] K. Inoue. Extended logic programs with default assumptions. In Koichi Furukawa, editor, *8th Int. Conf. on LP*, pages 490–504. MIT Press, 1991.

[JH91] Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the International Logic Programming Symposium*, pages 167–186. MIT Press, October 1991.

[JM92] Sverker Janson and Johan Montelius. Design of a Sequential Prototype Implementation of the Andorra Kernel Language. Sics research report, in preparation, Swedish Institute of Computer Science, 1992.

[Jon91] K. Jonker. On the semantics of conflit resolution in truth maintenance systems. Technical report, Univ. of Utrecht, 1991.

[Kas87] R. T. Kasper. Unification method for disjunctive feature descriptions. In *27th Annual Meeting of the Association for Computational Linguistics*, Standford, CA, 1987.

[Kow79] Robert A. Kowalski. *Logic for Problem Solving*. Elsevier North-Holland Inc., 1979.

[Kow90] R. Kowalski. Problems and promises of computational logic. In John Lloyd, editor, *Computational Logic*, pages 1–36. Basic Research Series, Springer–Verlag, 1990.

[KS90] R. Kowalski and F. Sadri. Logic programs with exceptions. In Warren and Szeredi, editors, *7th Int. Conf. on LP*. MIT Press, 1990.

[KT88] D. B. Kemp and R. W. Topor. Completeness of a top–d query evaluation procedure for stratified databases. In Kowalski and Bowen, editors, *Proc. of the Fifth Int. Conf. and Symposium on Logic Programming*, pages 178–194. ALP, MIT Press, 1988.

[LS93] V. Lifschitz and G. Schwarz. Extended logic programs as autoepistemic theories. In L. M. Pereira and A. Nerode, editors, *2nd Int. Ws. on LP & NMR*, pages 101–114. MIT Press, 1993.

[Mah88] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. Technical report, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, U.S.A., 1988.

[MK91] John T. Maxwell and Ronald M. Kaplan. A method for disjunctive constraint satisfaction. In Massaru Tomita, editor, *Current Issues in Parsing Technology.* Kluwer Academic Publishers, 1991.

[MT93] V. Marek and M. Truszczynski. Reflexive autoepistemic logic and logic programming. In L. M. Pereira and A. Nerode, editors, *2nd Int. Ws. on LP & NMR*, pages 115–131. MIT Press, 1993.

[Nai85] Lee Naish. *Negation and Control in Prolog.* Lecture notes in Computer Science 238. Springer–Verlag, 1985.

[PA92] L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In B. Neumann, editor, *European Conf. on AI*, pages 102–106. John Wiley & Sons, 1992.

[PA93a] L. M. Pereira and J. J. Alferes. Contradiction: when avoidance equal removal. Part II. In R. Dyckhoff, editor, *4th Int. Ws. on Extensions of LP*, pages 17–26. Univ. of St. Andrews, 1993.

[PA93b] L. M. Pereira and J. J. Alferes. Optative reasoning with scenario semantics. In D. S. Warren, editor, *10th Int. Conf. on LP*, pages 601–615. MIT Press, 1993.

[PAA91a] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Contradiction Removal within Well Founded Semantics. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *LP & NMR*, pages 105–119. MIT Press, 1991.

[PAA91b] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Counterfactual reasoning based on revising assumptions. In Ueda and Saraswat, editors, *Int. LP Symp.*, pages 566–577. MIT Press, 1991.

[PAA91c] L. M. Pereira, J. N. Aparício, and J. J. Alferes. A derivation procedure for extended stable models. In *Int. Joint Conf. on AI.* Morgan Kaufmann, 1991.

[PAA91d] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Nonmonotonic reasoning with well founded semantics. In Koichi Furukawa, editor, *8th Int. Conf. on LP*, pages 475–489. MIT Press, 1991.

[PAA92a] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Contradiction removal semantics with explicit negation. In *Applied Logic Conf.* Preproceedings by ILLC, Amsterdam, 1992. To appear in Springer–Verlag LNAI.

[PAA92b] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Logic programming for nonmonotonic reasoning. In *Applied Logic Conf.* Preproceedings by ILLC, Amsterdam, 1992. To appear in Springer–Verlag LNAI.

[PAA93] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Non–monotonic reasoning with logic programming. *Journal of LP. Special issue on Nonmonotonic reasoning*, 17(2), 1993.

[PDA93a] L. M. Pereira, C. Damásio, and J. J. Alferes. Debugging by diagnosing assumptions. In P. Fritzson and H. Nilsson, editors, *1st Int. Ws. on Automatic Algorithmic Debugging, AADEBUG'93.* Preproceedings by Linkoping Univ., 1993. To appear in Springer–Verlag LNCS.

[PDA93b] L. M. Pereira, C. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal. In L. M. Pereira and A. Nerode, editors, *2nd Int. Ws. on LP & NMR*, pages 316–330. MIT Press, 1993.

[PDA93c] L. M. Pereira, C. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal in logic programs. In M. Filgueiras and L. Damas, editors, *6th Portuguese AI Conf.* Springer–Verlag, 1993.

[PMCA86] Luís Moniz Pereira, Luís Monteiro, José Cunha, and Joaquim N. Aparício. Delta Prolog: a distributed backtracking extension with events. In Ehud Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 69–83. Springer-Verlag, 1986.

[PP79] L. M. Pereira and A. Porto. Intelligent Backtracking and Sidetracking in Horn Clause Programs - the Theory. Report 2/79, Departamento de Informatica, Universidade Nóva de Lisboa, October 1979.

[Prz89] T. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. In *8th Symp. on Principles of Database Systems.* ACM SIGACT-SIGMOD, 1989.

[Prz90] T. Przymusinski. Extended stable semantics for normal and disjunctive programs. In Warren and Szeredi, editors, *7th Int. Conf. on LP*, pages 459–477. MIT Press, 1990.

[Prz91a] T. Przymusinski. A semantics for disjunctive logic programs. In Loveland, Lobo, and Rajasekar, editors, *ILPS'91 Ws. in Disjunctive Logic Programs*, 1991.

[Prz91b] T. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424, 1991.

[Prz91c] T. Przymusinski. Stationary semantics for normal and disjunctive programs. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *2nd Int. Conf. on DOOD.* LNCS 566, Spriger–Verlag, 1991.

[Prz93] T. Przymusinski. Static semantics for normal and disjunctive programs. Technical report, Dep. of Computer Science, Univ. of California at Riverside, 1993.

[PW90] D. Pearce and G. Wagner. Reasoning with negative information I: Strong negation in logic programs. In L. Haaparanta, M. Kusch, and I. Niiniluoto, editors, *Language, Knowledge and Intentionality*, pages 430–453. Acta Philosophica Fennica 49, 1990.

[Ros92] K. Ross. A procedural semantics for well–founded negation in logic programs. *Journal of Logic Programming*, 13(1):1–22, 1992.

[Sak92] C. Sakama. Extended well–founded semantics for paraconsistent logic programs. In *Fifth Generation Computer Systems*, pages 592–599. ICOT, 1992.

[SC93] V. Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I.* PhD thesis, University of Bristol, August 1993.

[SCWY91a] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[SCWY91b] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 825–839. MIT Press, June 1991.

[SCWY91c] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting full Prolog on the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

[Sil93] Fernando M. A. Silva. *An Implementation of Or-Parallel Prolog on a Distributed Shared Memory Architecture.* PhD thesis, Dept. of Computer Science, Univ. of Manchester, September 1993.

[Smi91] Donald Smith. Constraint operations for clp($\mathcal{FT}$). In Koichi Furukawa, editor, *Logic Programming: Proceedings of the 8th International Conference.* MIT Press, 1991.

[Smo89] Gert Smolka. Feature constraint logics for unification grammars. Technical report, IBM Wissenschafliches Zentrum, Institut für Wissensbasierte Systeme, 1989. IWBS Report 93.

[SSM$^+$87] M. Sato, H. Shimizu, A. Matsumoto, K. Rokusawa, and A. Goto. KL1 Execution Model for PIM Cluster with Shared Memory. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 338–355. University of Melbourne, "MIT Press", May 1987.

[Tay90] A. Taylor. LIPS on a MIPS: Results from a Prolog Compiler for a RISC. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 174–185. MIT Press, June 1990.

[TB93] E. Tick and C. Banerjee. Performance evaluation of Monaco compiler and runtime kernel. In *ICLP93*, pages 757–773, 1993.

[TFar] Ana Paula Tomás and M. Filgueiras. Solving linear diophantine equations. *Reports of the Institute of Cybernetics, Tallinn*, to appear.

[TS86] H. Tamaki and T. Sato. Old resolution with tabulation. In *Proc. of 3rd International Conference on Logic Programming*, pages 84–98, 1986.

[VAMD91] Henk Van Acker, Remco Moolenaar, and Bart Demoen. A parallel implementation of AKL. Presented at the ILPS workshop on Parallel Logic Programming, October 1991.

[Van90] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.

[Wag91] G. Wagner. A database needs two kinds of negation. In B. Thalheim, J. Demetrovics, and H-D. Gerhardt, editors, *Mathematical Foundations of Database Systems*, pages 357–371. LNCS 495, Springer–Verlag, 1991.

[War83] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[War88] David H. D. Warren. The Andorra model. Presented at Gigalips Project workshop, University of Manchester, March 1988.

[War89] David H. D. Warren. Extended Andorra model. PEPMA Project workshop, University of Bristol, October 1989.

[Yan89] Rong Yang. Solving Simple Substitution Ciphers in Andorra-I. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 113–128. MIT Press, June 1989.

CONCURRENCY

Domain Independent Ask Approximation in CCP[*]

Enea Zaffanella
Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa
`zaffanel@di.unipi.it`

Abstract

The main difficulty in the definition of a static analysis framework for CC programs is probably related to the correct approximation of the entailment relation between constraints. This approximation is needed for the abstract evaluation of the ask guards and directly influences the overall precision of the analysis. In this paper we provide a solution to this problem by stating reasonable correctness conditions relating the abstract and the concrete domains of computation. The solution is domain independent in the sense that it can be applied to the class of *downward closed* observations. Properties falling in this class (e.g. freeness) have already been studied in the context of the analysis of sequential logic programs. We believe that the same abstract domains can be usefully applied to the CC context to provide meaningful ask approximations.

1 Introduction

Concurrent Constraint (CC) programming [16] arises as a generalization of both concurrent logic programming and constraint logic programming (CLP). In the CC framework processes are executed concurrently in a shared *store*, a constraint representing the global state of the computation. Communication is achieved by ask and tell basic actions. A process telling a constraint simply adds it to the current store, in a completely asynchronous way. Synchronization is achieved through *blocking asks*. Namely the process is suspended when the store does not entail the ask constraint and it remains suspended until the store entails it. While being elegant from a theoretical point of view, this synchronization mechanism turns out to be very difficult to model in the context of static analysis. The reason for such a problem lies in the anti–monotonic nature of the ask operator wrt the asked constraint: if we replace this constraint with a weaker one we obtain stronger observations. As a consequence, the approximation theory developed to correctly characterize *upward closed* (i.e. closed wrt entailment) properties becomes useless when we are looking for a domain independent solution to the ask approximation problem [18].

In this paper we thus consider the *downward closed* properties and we specify suitable domain independent correctness conditions that allow to overcome the problem of a safe

*This work has been supported by the "PARFORCE" (Parallel Formal Computing Environment) BRA-Esprit II Project n. 6707.

Progr	::=	Dec. Agent
Dec	::=	ϵ
	|	`p(x):-Agent. Dec`
Agent	::=	`Stop`
	|	`tell(`c`)`
	|	$\exists$ x in Agent
	|	Agent || Agent
	|	$\sum_{i=1}^{n}$ `ask(`c_i`)->Agent`$_i$
	|	`p(y)`

Table 1: The syntax

abstraction of ask constraints. In particular we develop an approximation theory that correctly detects the definite suspension of an ask guard. This information can be used in many ways, e.g. debugging of cc programs as well as identifying processes that are definitely serialized (so that we avoid their harmful parallel execution). However its usefulness is first of all in the improvement of the precision of the static analysis framework, as it allows to cut the branches of code that will not be considered in the concrete computation.

This (partial) classification of cc program's observations is not new. See [12] for an interesting discussion about *safety* and *liveness* properties, being downward closed and upward closed respectively. As a matter of fact, in the literature there already exist abstract domains developed for the static analysis of sequential (constraint) logic languages dealing with downward closed observations, e.g. freeness in the Herbrand as well as in arithmetic constraint systems [6]. It is our opinion that these same abstract domains can be usefully applied to the cc context and provide meaningful ask approximations.

2 The language

cc is not a language, it is a class of languages parametric wrt the underlying constraint system. In [16] constraint systems are defined by enclosing typical cylindric algebra's operators (cylindrifications and diagonal elements [10]) in the well known formalization of *partial information systems* [17], which model the gathering and the management of a set of elementary assertions by means of a compact entailment relation. We refer to [16] for a more detailed presentation.

Definition 2.1
A (cylindric) *constraint system* $C^\top = \langle\, C \cup \{false\}, \overset{\text{reverse of entailment}}{\dashv}, true, false, \otimes, \sqcap, V, \exists_x, d_{xy} \,\rangle$ is an algebraic structure where

- $\langle\, C, \dashv, true, \otimes, \sqcap \,\rangle$ is a partial information system
- *false* is the top element

Operational semantics à la Saraswat et al Page 91

- V is a denumerable set of variables
- $\forall x, y \in V$, $\forall c, d \in C$, the cylindric operator $\exists_x$ satisfies
  1. $\exists_x false = false$ — to implement local store
  2. $\exists_x c \dashv c$
  3. $c \dashv d$ implies $\exists_x c \dashv \exists_x d$
  4. $\exists_x(c \otimes \exists_x d) = \exists_x c \otimes \exists_x d$
  5. $\exists_x(\exists_y c) = \exists_y(\exists_x c)$ — to model parameter passing
- $\forall x, y, z \in V$, $\forall c \in C$, the diagonal element d_{xy} satisfies
  1. $d_{xx} = true$
  2. $z \neq x, y$ implies $d_{xy} = \exists_z(d_{xz} \otimes d_{zy})$
  3. $x \neq y$ implies $c \dashv d_{xy} \otimes \exists_x(c \otimes d_{xy})$

Note that we are distinguishing between the consistent constraints C and the top element *false* representing inconsistency. In the following we will write $\mathcal{C}$ to denote the subalgebra of consistent constraints, namely the set C together with the constraint system's operators restricted to work on C. We will denote operators and their restrictions in the same way and we will often refer to $\mathcal{C}$ as a "constraint system".

Tables 1 and 2 introduce the syntax and the operational semantics of cc languages. For notational convenience, we consider processes having one variable only in the head. We also assume that for all the procedure names occurring in the program text there is a corresponding definition. The operational model is described by a transition system $T = (Conf, \longrightarrow)$. Elements of *Conf* (configurations) consist of an agent and a constraint, representing the residual computation and the global store respectively. $\longrightarrow$ is the (minimal) transition relation satisfying axioms **R1**-**R5**.

The execution of an elementary tell action simply adds the constraint c to the current store d (no consistency check). Axiom **R2** describes the hiding operator. The syntax is extended to deal with a local store c holding information about the hidden variable x. Hence the information about x produced by the external environment does not affect the process behaviour and conversely the external environment cannot access the local store. Initially the local store is empty, i.e. $\exists$ x in $A \equiv \exists(x, true)$ in A. Parallelism is modeled as *interleaving* of basic actions. In a guarded choice operator, a branch A_i is enabled in the current store d iff the corresponding guard constraint ask(c_i) is entailed by the store, i.e. $d \vdash c_i$. The guarded choice operator nondeterministically selects one enabled branch A_i and behaves like it. If there is no enabled branch then it suspends, waiting for other processes to add the desired information to the store. Finally, when executing a procedure call, rule **R5** models parameter passing without variable renaming [16], where `p(x):-`$A \in P$ and $\Delta^{\mathtt{y}}_{\mathtt{x}} A$ is defined as follows [5].

$$\Delta^{\mathtt{y}}_{\mathtt{x}} A \;=\; \begin{cases} A & \text{if } \mathtt{x} \equiv \mathtt{y} \\ \exists\,\mathtt{x}\ \mathtt{in}(\mathtt{tell}(d_{xy}) \parallel A) & \text{otherwise} \end{cases}$$

A *c*-computation s for a program $D.A$ is a possibly infinite and fair sequence of configurations $\langle\, A_i, c_i \,\rangle_{i<\omega}$ such that $A_0 = A$ and $c_0 = c$ and for all $i < |s|$, $\langle\, A_i, c_i \,\rangle \longrightarrow \langle\, A_{i+1}, c_{i+1} \,\rangle$.

R1	$\langle\, \texttt{tell}(c), d \,\rangle \longrightarrow \langle\, \texttt{Stop}, d \otimes c \,\rangle$
R2	$\dfrac{\langle\, A, c \otimes \exists_x d \,\rangle \longrightarrow \langle\, A', c' \,\rangle}{\langle\, \exists(x,c)\,\texttt{in}\,A, d \,\rangle \longrightarrow \langle\, \exists(x,c')\,\texttt{in}\,A', d \otimes \exists_x c' \,\rangle}$
R3	$\dfrac{\langle\, A, c \,\rangle \longrightarrow \langle\, A', d \,\rangle}{\begin{array}{c}\langle\, A \parallel B, c \,\rangle \longrightarrow \langle\, A' \parallel B, d \,\rangle \\ \langle\, B \parallel A, c \,\rangle \longrightarrow \langle\, B \parallel A', d \,\rangle\end{array}}$
R4	$\dfrac{j \in \{1, \ldots, n\} \ \wedge \ d \vdash c_j}{\langle\, \sum_{i=1}^{n} \texttt{ask}(c_i)\texttt{->}A_i, d \,\rangle \longrightarrow \langle\, A_j, d \,\rangle}$
R5	$\dfrac{\texttt{p(x):-}A \in P}{\langle\, \texttt{p(y)}, d \,\rangle \longrightarrow \langle\, \Delta_{\mathtt{x}}^{\mathtt{y}} A, d \,\rangle}$

Table 2: The transition system T

Let $\not\longrightarrow$ denote the absence of admissible transitions. Computations reaching configuration $\langle\, A_n, c_n \,\rangle \not\longrightarrow$ are called *finite* computations and c_n is the (finite) computed answer constraint. If the residual agent A_n contains some choice operators then the corresponding computation is *suspended*, otherwise it is a *successful* computation and in this case we denote A_n by ϵ.

Definition 2.2 The semantics for program $P = D.A$ in the store c is

$$\begin{array}{rcl}\mathcal{O}[\![D.A]\!](c) & = & \left\{ d \in C \;\middle|\; \langle\, A\,, c \,\rangle \overset{*}{\longrightarrow} \langle\, B\,, d \,\rangle \not\longrightarrow \right\} \\ & \cup & \left\{ d \in C \;\middle|\; \begin{array}{l} \langle\, A_0\,, c_0 \,\rangle \longrightarrow \ldots \longrightarrow \langle\, A_i\,, c_i \,\rangle \longrightarrow \ldots \\ A_0 = A\,,\ c_0 = c\,,\ d = c_0 \otimes \ldots \otimes c_i \otimes \ldots \end{array} \right\}\end{array}$$

Note that this semantics collects the limit constraints of infinite computations as well as the answer constraints associated to finite computations, regardless of whether the latter are successful or suspended. In any case we are considering consistent constraints only, i.e. we disregard all computations delivering *false*.

3 Program properties and approximations

As we have seen, the operational semantics of a CC program associates each initial store c to the set of all the consistent constraints that we obtain by executing $P = D.A$ at c. In a similar way we define a *semantic property* ϕ as a subset of C, namely the set of consistent constraints that satisfy the property. Therefore a program satisfies a semantic property ϕ at c iff the observations of the program are a *subset* of the property, i.e. $\mathcal{O}[\![P]\!](c) \subseteq \phi$. Following this general view, the static analysis of a CC program can be formalized as a finite construction of an approximation (a superset) of the program

denotation. If the approximation satisfies the semantic property, then we can correctly say that our program satisfies the property too. Abstract interpretation [3] formalizes the approximation construction process by mapping concrete semantic objects and operators into corresponding abstract semantic objects and operators.

We write $\uparrow(\phi)$ to denote the *upward closure* of the program property ϕ, namely the set $\{c \in C \mid \exists\, d \in \phi \,.\, c \vdash d\}$; a property is upward closed iff it is equivalent to its upward closure, i.e. $\phi = \uparrow(\phi)$. Downward closed properties are defined dually. As an example, consider the Herbrand constraint system C_H. If the constraint $c \in C_H$ binds variable x to a ground term, then all the constraints $d \in C_H$ such that $d \vdash c$ will bind x to a ground term; therefore *groundness* is an upward closed property. On the other hand, *freeness* is a downward closed property. A variable x is free in $c \in C_H$ iff there does not exist a term functor f/n such that $c \vdash (\exists_{y_1} \ldots \exists_{y_n}\, x = f(y_1, \ldots, y_n))$. Thus, if x is free in c then it will be free in all the constraints $d \in C_H$ such that $c \vdash d$. However, there obviously exist properties falling in none of these two classes, e.g. *independence*. Let us say that variables x and y share in $c \in C_H$ iff c binds x and y to the terms t_x and t_y such that $\mathit{var}(t_x) \cap \mathit{var}(t_y) \neq \emptyset$. Variables x and y are independent in c if they do not share in c. Now, if x and y share in c, we can choose constraints $d_1, d_2 \in C_H$ such that $d_1 \vdash c \vdash d_2$ and such that x and y are independent in both d_1 and d_2.

Ordering closed properties are very common in the static analysis of logic languages and furthermore they are easier to verify, because correctness of the abstract interpretation can be based on a semantics returning ordering closed observations. In [18] entailment closed[1] properties are considered. The main result is that it is impossible to develop a meaningful generalized semantics for CC languages in the style of [9], namely the only way to correctly abstract ask constraints in a domain independent fashion is a trivial approximation.

In this work we turn our interest upon downward closed properties and we show that a (carefully chosen but natural) notion of correctness of the abstract domain wrt the concrete one allows to automatically derive a correct approximation of all the asks occurring in the program. Dealing with such a class of properties, the collecting semantics can be defined naturally as the downward closure of the operational semantics, as there is no benefit in considering a stronger one [18].

Remark 3.1 *If ϕ is downward closed then $\mathcal{O}[\![P]\!](c) \subseteq \phi \;\Leftrightarrow\; \downarrow(\mathcal{O}[\![D]\!](c)) \subseteq \phi$.*

As we are observing infinite computations also, we have to be careful when defining the downward closed properties that we are interested in. In particular we have to remember that usually the correctness of our abstract semantic construction is based on the Scott's induction principle; this principle is only valid for *admissible* properties.

Definition 3.1 A property $\phi \subseteq C$ is *admissible* iff ϕ is closed under directed *lub*'s.

This definition means that whenever an admissible property is satisfied by all the finite approximations of the semantics, then *the* semantics will satisfy the property too. As an

[1]Due to a dual definition of the ordering on the constraint system, in [18] entailment closed properties are the downward closed ones. The choice of turning the domain upside–down was influenced by the standard theory of semantic approximation by means of upper Galois insertions [3].

example of a property that is not admissible, consider the following definition of *nongroundness*: a variable x is nonground in $c \in C_H$ iff c binds x to a term t such that $var(t) \neq \emptyset$. Given the infinite chain of constraints $c_i \equiv (\exists_y\, x = f^i(y)) \in C_H$, for every $i < \omega$ we have that x is nonground in c_i. However, considering the limit constraint $c \equiv \bigotimes_{i<\omega} c_i = (x = f^\omega)$ one observes that x is *not* nonground in c. In order to grant the correctness of this analysis, we have to redefine the property, e.g. by stating that if c binds a x to an infinite term then x is nonground in c.

Hence, in this work we are interested in downward closed and admissible program properties. The Hoare's powerdomain [14, 17] construction over the constraint system characterizes this kind of observations.

Definition 3.2 The Hoare's powerdomain of the constraint system $\mathcal{C}$ is the complete lattice $\mathcal{H}(\mathcal{C}) = \langle \mathcal{P}{\downarrow}(C), \subseteq, \{true\}, C, \uplus, \cap \rangle$, where $\mathcal{P}{\downarrow}(C)$ is the set of all the nonempty, downward closed and admissible subsets of C; $\uplus$ is the closure under directed C-lub's of the set theoretical union; $:\{\cdot\}: \; : C \to \mathcal{P}{\downarrow}(C)$ defined as $:\{c\}: = {\downarrow}\{c\}$ is the singleton embedding function.

The alert reader would observe that this collecting semantics models nonempty observations only. From a semantic construction point of view, this is not completely satisfactory as we cannot describe the behaviour of a program having inconsistent computations only. However, the alternative choice of considering failed computations also would imply some negative consequences. Firstly, it would complicate the formalization of the correctness conditions, requiring a special treatment for inconsistency. Moreover it would degrade the precision of our static analysis, adding very little to the understanding of the program. To see this, observe that when considering downward closed observations a failed computation has to be interpreted as "the program *may* fail", meaning that anything can happen. Also consider that there are CC languages explicitly designed to statically avoid the possibility of a failing computation (see [15] for a discussion of this topic in distributed programming).
From now on $\tilde{\otimes}$ and $\tilde{\exists}_x$ will denote the extensions of $\otimes$ and $\exists_x$ over $\mathcal{H}(\mathcal{C})$.

- $\forall S, T \in \mathcal{P}{\downarrow}(C)\,.\; S \,\tilde{\otimes}\, T = \biguplus \Big\{ :\{c \otimes d\}: \;\Big|\; c \in S,\, d \in T,\, c \otimes d \in C \Big\}$
- $\forall S \in \mathcal{P}{\downarrow}(C)\,.\; \tilde{\exists}_x S = \biguplus \Big\{ :\{\exists_x c\}: \;\Big|\; c \in S \Big\}$

Note that the *merge over all paths* operator [3] is provided by the *lub* of $\mathcal{H}(\mathcal{C})$. Also note that in general $\tilde{\otimes}$ is not idempotent, while being extensive.

4 Correctness

In this section we formalize the notion of *correctness* of an abstract domain wrt a concrete constraint system when downward closed properties are observed. As outlined in the previous section, we have to grant the existence of an upper Galois insertion relating the Hoare's powerdomain of the concrete constraint system and the abstract domain of descriptions, together with suitable correctness conditions regarding the domain's operators.

Definition 4.1 An abstract domain $\mathcal{A} = \langle L, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp, \sqcup^\sharp, \sqcap^\sharp, \otimes^\sharp, V, \exists^\sharp{}_x, d^\sharp{}_{xy} \rangle$ is *down–correct* wrt the constraint system $\mathcal{C} = \langle C, \dashv, true, \otimes, \sqcap, V, \exists_x, d_{xy} \rangle$ using α iff $\forall S, T \in \mathcal{P}{\downarrow}(C)$, $\forall x, y \in V$

1. $\mathcal{L} = \langle L, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp, \sqcup^\sharp, \sqcap^\sharp \rangle$ is a complete lattice
2. there exists γ s.t. (α, γ) is an upper Galois insertion[2] relating $\mathcal{H}(\mathcal{C})$ and $\mathcal{A}$.
3. $\alpha(S \,\tilde{\otimes}\, T) \sqsubseteq^\sharp \alpha(S) \otimes^\sharp \alpha(T)$
4. $\alpha(\tilde{\exists}_x S) \sqsubseteq^\sharp \exists^\sharp{}_x \alpha(S)$
5. $\alpha(:\{d_{xy}\}:) \sqsubseteq^\sharp d^\sharp{}_{xy}$

From now on, we assume that the abstract domain $\mathcal{A}$ is *down–correct* wrt the constraint system $\mathcal{C}$ using α and prove that such a notion of correctness implies the correctness of any abstract semantic construction based on the abstract interpretation theory. This means that the proof is valid for any abstract semantics that systematically mimics the basic concrete semantic operators ($\uplus$, $\otimes$, $\exists_x$, d_{xy}) and the relation $\dashv$ by using the corresponding abstract operators ($\sqcup^\sharp$, $\otimes^\sharp$, $\exists^\sharp{}_x$, $d^\sharp{}_{xy}$) and the relation $\sqsubseteq^\sharp$. To this end it is sufficient to consider the operational semantics.

Definition 4.2 Given the concrete agent A, the corresponding abstract agent $A^\sharp = \alpha(A)$ is obtained by replacing all the concrete constraints $c \in C$ occurring in A by the corresponding abstractions $c^\sharp = \alpha(:\{c\}:) \in L$.

The following lemma shows that the abstract program correctly mimics each transition of the concrete one. This also means that if the abstract program suspends, then the concrete program suspends too. Let A be an agent defined over the constraint system $\mathcal{C}$, let $c \in C$ be a concrete store and let $c^\sharp \in L$ be a description such that $\alpha(:\{c\}:) \sqsubseteq^\sharp c^\sharp$.

Lemma 4.1 (correctness)
$\langle A\,,\, c \rangle \longrightarrow \langle B\,,\, d \rangle$ *implies* $\langle \alpha(A)\,,\, c^\sharp \rangle \longrightarrow \langle \alpha(B)\,,\, d^\sharp \rangle$ *and* $\alpha(:\{d\}:) \sqsubseteq^\sharp d^\sharp$.

The following proposition is proved by induction on the number of transitions.

Proposition 4.2 *For every concrete c-computation of P yielding the constraint $d \in C$ there exists a corresponding abstract $\alpha(:\{c\}:)$-computation of $\alpha(P)$ yielding the description $d^\sharp$ such that $\alpha(:\{d\}:) \sqsubseteq^\sharp d^\sharp$.*

Note that in general the converse of Lemma 4.1 does not hold. In particular the concrete program may suspend while the abstract one has a transition; as a consequence, a finite concrete computation can be mapped into a corresponding abstract infinite computation. Therefore, even in the case that we are interested in finite computations only, the abstract semantics *must* consider infinite computations in order to be correct.

[2]Given two complete lattices $\langle L, \leq \rangle$ and $\langle L', \leq' \rangle$, an *upper Galois connection* between L and L' is a pair of adjoint functions (α, γ) such that $\alpha : L \to L'$ and $\gamma : L' \to L$ and $\forall x \in L\,.\,\forall y \in L'\,.\,\alpha(x) \leq' y \Leftrightarrow x \leq \gamma(y)$. An upper Galois *insertion* between L and L' is an upper Galois connection such that α is surjective (equivalently, γ is one-to-one).

Definition 4.2 does not require that the abstract domain is a constraint system and neither that it can be obtained as the Hoare's powerdomain of a constraint system. In the latter case we are in an *ideal situation* where a simpler notion of correctness can be used instead.

Definition 4.3
An abstract constraint system $\mathcal{A} = \langle L, \dashv^\sharp, \bot^\sharp, \top^\sharp, \otimes^\sharp, \sqcap^\sharp, V, \exists^\sharp{}_x, d^\sharp{}_{xy} \rangle$ is *correct* wrt the constraint system $\mathcal{C} = \langle C, \dashv, true, \otimes, \sqcap, V, \exists_x, d_{xy} \rangle$, using a surjective and monotonic function $\alpha : C \to D$, iff for each $c, d \in C$, $x, y \in V$

1. $\alpha(c \otimes d) \dashv^\sharp \alpha(c) \otimes^\sharp \alpha(d)$
2. $\alpha(\exists_x c) \dashv^\sharp \exists^\sharp{}_x \alpha(c)$
3. $\alpha(d_{xy}) = d^\sharp{}_{xy}$

Let $\mathcal{A}$ be an abstract constraint system which is correct wrt the constraint system $\mathcal{C}$ using α. Observe that $\otimes^\sharp$ is the *lub* over $\mathcal{A}$.

Proposition 4.3
1. $\mathcal{H}(\mathcal{A})$ is down–correct *wrt $\mathcal{H}(\mathcal{C})$ using $\tilde{\alpha}$ (the additive extension of α)*
2. α is a complete $\otimes$–morphism between C and L
3. $\tilde{\alpha}$ is a complete $\tilde{\otimes}$–morphism between $\mathcal{P}\!\downarrow\!(C)$ and $\mathcal{P}\!\downarrow\!(L)$

Defining abstract domains based on correct abstract constraint systems is a very difficult task. The previous proposition gives an explanation of this assertion: these domains have to satisfy properties that usually are too strong.

4.1 A toy example

As a first example we present the abstract constraint system of *untouched variables*[3] $\mathcal{V} = \langle \mathcal{P}(V), \subseteq, \emptyset, V, \otimes', \cap, V, \exists'_x, d'_{xy} \rangle$, where

$$\begin{aligned} S \otimes' T &= S \cup T \\ \exists'_x S &= S \backslash \{x\} \end{aligned} \qquad\qquad d'_{xy} = \begin{cases} \{x, y\} & \text{if } x \not\equiv y \\ \emptyset & \text{otw.} \end{cases}$$

Let us assume that $\mathcal{C}$ is a concrete constraint system having variables in V and satisfying the following axiom [5]: $\forall c, d \in C \;.\; \exists_x c \vdash d \;\Rightarrow\; \exists_x d = d$. Note that even if this axiom is not a consequence of Definition 2.1, it is true in almost all the "real" constraint systems.

Proposition 4.4 *Let $\alpha : C \to \mathcal{P}(V)$ being defined as $\alpha(c) = \{x \in V \mid \exists_x c \neq c\}$. The abstract constraint system $\mathcal{V}$ is correct wrt $\mathcal{C}$ by using α.*

[3]To our knowledge, this domain has been firstly introduced in [8]. The formal definition of α was given to me by Catuscia Palamidessi, during an interesting discussion related to other topics.

Therefore, we just are in the ideal situation of Definition 4.3 and we can define our abstract domain as the Hoare's powerdomain of $\mathcal{V}$. Having proved correctness, we can approximate every concrete ask evaluation (i.e. entailment check) by the corresponding abstract ask evaluation. Let us see the intuition behind this result. Suppose the abstract ask evaluation does not succeed; this means that there exists a variable x occurring free in the concrete ask constraint such that x is definitely unbounded in all the concrete constraints described by the abstract store. As a consequence all the associated concrete computations will suspend too and we are safe.

4.2 Abstracting the constraint system $\mathcal{R}_{LinEq}$

Previous example seems just a toy. However, the same approach is valid for any admissible downward closed property of any constraint system. Some examples of this kind of abstract domains can be found in the literature.

[6] describes an abstract domain for the static analysis of CLP programs that is useful for the detection of *definitely free* variables in the presence of both Herbrand constraints as well as systems of linear equations. Let us consider the latter case. Given a linear equation system

$$E = \left\{ \begin{array}{llllllll} a_{11}X_1 & + & a_{12}X_2 & + \ldots + & a_{1n}X_n & = b_1 \\ \ldots & & \ldots & \ldots & \ldots & \ldots \\ a_{m1}X_1 & + & a_{m2}X_2 & + \ldots + & a_{mn}X_n & = b_m \end{array} \right.$$

where $X_1, \ldots, X_n$ are variables and a_{ij} and b_j are numbers, variable X_i is definitely free if there does not exist a linear combination of the equations in E having the form $X_i = n$. Denoting $lc(E)$ the infinite set of linear combinations of equations in E, they define the following abstraction function.

$$\alpha(E) = \left\{ \{X_1, \ldots, X_k\} \;\middle|\; \begin{array}{l} (a_1 X_1 + \ldots + k X_k = b) \in lc(E), \\ a_i \neq 0 \;\; i = 1, \ldots, k \end{array} \right\}.$$

We refer to [6] for a complete definition of the domain and of the abstract operators. Intuitively, the correctness of the analysis ensures that all the possible linear combinations of concrete equations are described by the computed abstract element. As a particular case, if the abstract linear combination $\{X_i\}$ is not a member of the abstract store description, we can safely say that variable X_i is free. [6] also shows how to correctly deal with inequalities and disequations (i.e. the constraint system $\mathcal{R}^{\neq}_{Lin}$ is considered).

5 Toward an abstract semantics

In this section we will informally consider the problems related to the construction of an abstract semantics that correctly approximates the standard one in the case of downward closed observations.

In the general case, the observations of a CC program are not invariant wrt different schedulings of parallel processes, i.e. the operational semantics is not confluent. In principle, confluence is not needed to correctly define a static analysis framework. However, in order to be really useful, a static analysis must be correct wrt all the possible scheduling and *must not be* too inefficient. Therefore, when considering programs being a little bit bigger than toy examples, confluence becomes as desirable as correctness [8]. As a matter of fact, almost all the literature concerning the static analysis of CC languages considers non–standard semantics that are confluent [1, 2, 7, 8, 18]. These semantics are correct wrt the standard one, but usually *must pay* in terms of accuracy of the results.

This is not the case when considering downward closed properties, because we can base our static analysis on a confluent semantics being as precise as the standard one. Confluence is easily obtained by reading the CC indeterministic program as if it were an *angelic* program [11], that is by interpreting all the *don't care* choice operators of the program as *don't know* choice operators. In the angelic case, when considering a choice operator we split the control and consider all the branches. In the operational semantics this difference is captured by replacing rule R4 in Table 2 with the following.

$$\text{R4}' \quad \frac{d \vdash c}{\langle\, \texttt{ask}(c)\texttt{->}A, d\,\rangle \longrightarrow \langle\, A, d\,\rangle} \qquad\qquad \text{R4}'' \quad \frac{j \in \{1, \ldots, n\}}{\langle\, \sum_{i=1}^{n} A_i, d\,\rangle \longrightarrow \langle\, A_j, d\,\rangle}$$

Observe that the only difference between the two programs is that the original program has less suspensions; however, due to the monotonic nature of CC computations, for every suspended computation of the angelic program there exists a (terminated or suspended or infinite) computation in the original program that computes a stronger store. Let $\mathcal{O}'$ be the operational semantics based on the confluent transition system.

Proposition 5.1 *For all* $c \in C$. $\downarrow(\mathcal{O}[\![P]\!](c)) = \downarrow(\mathcal{O}'[\![P]\!](c))$.

Thus a first proposal of an abstract semantic construction can be based on the confluent transition system operational semantics. Technical problems related to termination can be solved essentially in the same way as it was done in [1].

In [16] it is shown how to elegantly model a deterministic CC process as an upper closure operator (uco), i.e. a monotonic, extensive and idempotent function over the constraint system. The main property of this kind of representation is that any uco is fully determined by the set of its fixpoints. Moreover all the semantic operators on processes are naturally mapped into simple set theoretic operators over their representations, e.g. the parallel composition of two processes is obtained by taking the intersection of their fixpoints' sets. [11] study the extension of such a semantics on angelic CC languages, where only local choice operators are allowed and upward closed observations are considered.

If the abstract domain we are dealing with is based on an abstract constraint system (see Definition 4.3) we are in a position to develop a semantic construction very similar to the latter. It is worth noting that, in such a semantic construction, the *process restartability* property is assumed. This property holds for deterministic programs [16] and it also holds for angelic programs when we consider upward closed observations [11], but it does not hold in the general case. However, when considering downward closed properties, it can be proved that correctness is still granted, while we pay something in the approximation's precision.

Unfortunately, many interesting abstract domains modelling downward closed properties are not constraint systems. In these cases, if we are interested in a denotational abstract semantic construction, we can consider a suitable variant of the approach based on ask/tell traces developed in [4]. Here the first problem to solve is termination, because

a trace can be infinite even if defined over a finite abstract domain. We think that a notion of *canonical form* for traces (similar to the one developed in [16]) would suffice.

It is worth pointing out that the approximation theory developed in this work can be applied to any kind of semantic construction dealing with the basic mechanism of blocking ask. Therefore, even if all the semantics mentioned above only observe *the results* of a CC program, our technique can be also applied to semantics observing *the way* these results are actually computed. As an example, if we consider the *true concurrency* semantics developed in [13], the definite suspension information could be useful to obtain upper bounds to the degree of parallelism of a program or to discover undesired data dependencies between concurrent processes.

6 Conclusions and related works

The static analysis of CC languages is a relatively new but very active area of research. To our knowledge, this is the first work on this topic in which it is identified a domain independent correct approximation of ask constraints. Almost all the previous works about the static analysis of CC programs [1, 7, 8, 18] either consider a specific constraint system or *assume* that a correct ask approximation has already been found. In [2] a different kind of domain independent ask approximation has been considered. In our opinion, however, this framework requires the satisfaction of too strong correctness conditions and cannot be widely used.

The approximation described in the current work allows to detect definitely suspended branches of the computation and it may be therefore useful in the debugging and specialization of CC programs. It can be applied to a wide class of program properties, namely the downward closed ones. Some property falling in this class (e.g. freeness) has already been studied in the context of the static analysis of sequential (constraint) logic languages. In our opinion the same abstract domains can be used in the CC case, provided that a suitable semantic construction is identified. At the same time, we strongly believe that such a general result can motivate the study of "new" downward closed properties.

The definition of a suitable abstract semantics for the static analysis of this class of properties is an open problem. We have shown that if we are interested in downward closed properties only then we can assume that all the choice operators in our program are local, achieving the confluence of the computation without any loss of precision. In our opinion, however, an extensive study of the cost/precision tradeoffs of the different abstract semantics proposals is strongly needed.

References

[1] M. Codish, M. Falaschi, K. Marriott, and W. Winsborough. Efficient Analysis of Concurrent Constraint Logic Programs. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Proc. of the 20th International Colloquium on Automata, Languages, and Programming*, volume 700 of *Lecture Notes in Computer Science*, pages 633–644, 1993.

[2] C. Codognet and P. Codognet. A general semantics for Concurrent Constraint Languages and their Abstract Interpretation. In M. Meyer, editor, *Workshop on Constraint Processing at the International Congress on Computer Systems and Applied Mathematics, CSAM'93*, 1993.

[3] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.

[4] F.S. de Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In S. Abramsky and T. Maibaum, editors, *Proc. TAPSOFT'91*, volume 493 of *Lecture Notes in Computer Science*, pages 296–319. Springer-Verlag, Berlin, 1991.

[5] F.S. de Boer, C. Palamidessi, and A. Di Pierro. Infinite Computations in Nondeterministic Constraint Programming. *Theoretical Computer Science*. To appear.

[6] V. Dumortier, G. Janssens, M. Bruynooghe, and M. Codish. Freeness analysis in the presence of numerical constraints. In D. S. Warren, editor, *Proc. Tenth Int'l Conf. on Logic Programming*, pages 100–115. The MIT Press, Cambridge, Mass., 1993.

[7] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Compositional Analysis for Concurrent Constraint Programming. In *Proc. of the Eight Annual IEEE Symposium on Logic in Computer Science*, pages 210–221. IEEE Computer Society Press, 1993.

[8] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence and Concurrent Constraint Programming. In *Proc. of the Fourth International Conference on Algebraic Methodology and Software Technology (AMAST'95)*, Montreal, Canada, 1995.

[9] R. Giacobazzi, S. K. Debray, and G. Levi. A Generalized Semantics for Constraint Logic Programs. In *Proc. of the International Conference on Fifth Generation Computer Systems 1992*, pages 581–591, 1992.

[10] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras. Part I and II.* North-Holland, Amsterdam, 1971.

[11] R. Jagadeesan, V. Shanbhogue, and V. Saraswat. Angelic non-determinism in concurrent constraint programming. Technical report, System Science Lab., Xerox PARC, 1991.

[12] M. Z. Kwiatkowska. Infinite Behaviour and Fairness in Concurrent Constraint Programming. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 348–383, Beekbergen The Netherlands, June 1992. REX Workshop, Springer-Verlag, Berlin.

[13] U. Montanari and F. Rossi. Contextual Occurrence Nets and Concurrent Constraint Programming. In *Proc. Dagstuhl Seminar on Graph Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1994.

[14] G.D. Plotkin. Pisa lecture notes. Unpublished notes, 1981-82.

[15] V. A. Saraswat, K. Kahn, and J. Levy. Janus: A step towards distributed constraint programming. In S. K. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 431–446. The MIT Press, Cambridge, Mass., 1990.

[16] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundation of Concurrent Constraint Programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*, pages 333–353. ACM, 1991.

[17] D. Scott. Domains for Denotational Semantics. In M. Nielsen and E. M. Schmidt, editors, *Proc. Ninth Int. Coll. on Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, Berlin, 1982.

✓ [18] E. Zaffanella, G. Levi, and R. Giacobazzi. Abstracting Synchronization in Concurrent Constraint Programming. In M. Hermenegildo and J. Penjam, editors, *Proc. Sixth Int'l Symp. on Programming Language Implementation and Logic Programming*, volume 844 of *Lecture Notes in Computer Science*, pages 57–72. Springer-Verlag, 1994.

Modeling Real-Time in Concurrent Constraint Programming

F.S. de Boer[*] M. Gabbrielli[§]

Abstract

We develop a language for real-time programming based on the concurrent constraint programming (*ccp*) paradigm. The language, called *tccp*, is obtained by a natural timed interpretation of the usual *ccp* constructs and by the addition of a simple construct which allows one to specify timing constraints. We define the operational semantics of *tccp* via a transition system and introduce a compositional and fully abstract model based on timed reactive sequences.

1 Introduction

In the actual practice of programming many applications are time-critical. Examples of such applications are real-time process controllers and signal-processing systems. In general, time-critical applications require a programmer to specify the interaction with an environment given some timing constraints such as that a certain input is required within a certain bounded period of time. The resulting systems, usually called reactive, need then suitable programming languages which allow for the definition of timing primitives.

Concurrent synchronous languages such as ESTEREL [2], LUSTRE [6], SIGNAL [9] and Statecharts [7] have been specifically designed for reactive systems. These languages are based on the *instantaneous reaction* (or perfect synchrony) hypothesis: A program is activated by some *input signals* and reacts *instantly* by producing the required output. So computation is performed in no time, unless a statement which explicitly consumes time is present. Communication is done by instantaneous broadcasting to all the processes of the system and the presence or absence of a signal can be detected at any instant. The perfect synchrony assumption can be realized in practice by compiling pure programs (i.e. programs operating only on signals) into finite state automata whose single step execution time is bounded. A direct compilation of pure ESTEREL programs in hardware has also been defined.

The perfect synchrony hypothesis, even though natural from the user point of view, *conflicts* with the inherent temporality of physical processes. As a consequence temporal paradoxes arise, for example, in the form of programs which require a signal to be present iff it is not present. To solve this conflict, Saraswat et al. [13, 14] have proposed an integration of the asynchronous computational model of concurrent constraint programming (ccp) [11, 12, 15] with ideas from synchronous languages. The resulting languages, called timed concurrent constraint programming (`tcc`) and `default tcc`, are designed around the hypothesis of *bounded asynchrony*: Computation takes a bounded period of time rather than

[*]Universiteit Utrecht. Utrecht, The Netherlands. `frankb@cs.ruu.nl`.

[§]Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy. `gabbri@di.unipi.it`.

being instantaneous. The whole system evolves in cycles corresponding to time intervals, and each time interval is identified with the time needed for a ccp process to terminate a computation. Special primitives are added to the standard ccp constructs to control the temporal evolution of the system. In particular, the programmer has to transfer explicitly the (positive) information from a time instant to the next one using these temporal primitives.

In this paper, analogously to the case of (default) tcc, we use ccp as the starting language and we assume that computation takes a bounded period of time. However, differently from [13, 14], we introduce directly a timed interpretation of the usual programming constructs of ccp by considering the primitive ccp constructs ask and tell as the elementary actions whose evaluation take one time unit. Thus, in our model, each time interval is identified with the time needed for the underlying constraint system to accumulate the tell's and to answer the queries (ask's) issued at each computation step by the processes of the system. Then we use this interpretation as a basis for the introduction of a construct which allows one to specify timing constraints. As we discuss later, our approach requires a smooth extension of ccp. In particular, we do not require explicit transfer of information across time boundaries and we can use the usual ccp definitions for hiding and recursion. We describe semantically our timed extension of ccp both operationally, in terms of a transition system, and denotationally. The denotational semantics is based on sequences of pairs of constraints, so called reactive sequences, as in the untimed case. However these reactive sequences are now provided with a different interpretation which accounts for the timing aspects. Our main result shows that the denotational semantics is correct and fully abstract with respect to the operational semantics. This paper is organized as follows. In the next section we introduce our timed extension of ccp and its operational semantics. Section 3 describes how to derive some typical real-time constructs form the basic combinators of the language. In section 4 we define the denotational semantics and we state the full abstraction result. Finally, Section 5 concludes by comparing our approach to the existing timed extensions of ccp and by giving some directions for future research.

2 The language

In this section we first introduce the *tccp* language and provide its basic operational intuitions. Then we define formally the operational semantics of *tccp* using a transition system. As in [13, 14] the starting point is ccp, so we introduce first some basic notions related to this programming paradigm. We refer to [12, 15] for more details. The ccp languages are defined parametrically wrt to a given *constraint system.* The notion of cylindric constraint system has been formalized in [12] following Scott's treatment of information systems [16] and using ideas from cylindric algebras [8] in order to treat the hiding operator of the language in terms of a general notion of existential quantifier. Here we only consider the resulting structure.

Definition 2.1 Let $\langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false}\rangle$ be a complete algebraic lattice where $\sqcup$ is the lub operation, and *true*, *false* are the least and the greatest elements of $\mathcal{C}$, respectively. Assume given a (denumerable) set of variables *Var* with typical elements $x, y, z \ldots$. For each $x \in \mathit{Var}$ it is defined a function $\exists_x : \mathcal{C} \to \mathcal{C}$ such that, for any $c, d \in \mathcal{C}$:

(i) $c \vdash \exists_x(c)$, $\qquad$ (ii) if $c \vdash d$ then $\exists_x(c) \vdash \exists_x(d)$,
(iii) $\exists_x(c \sqcup \exists_x(d)) = \exists_x(c) \sqcup \exists_x(d)$, $\qquad$ (iv) $\exists_x(\exists_y(c)) = \exists_y(\exists_x(c))$.

Then $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false}, \mathit{Var}, \exists\rangle$ is a *cylindric constraint system.*

Following the standard terminology and notation, instead of $\leq$ we will refer to its inverse relation, denoted by $\vdash$ and called *entailment.* Formally, $\forall c, d \in \mathcal{C}.\ \ c \vdash d \ \Leftrightarrow\ d \leq c$. Moreover, in the sequel we will identify a system $\mathbf{C}$ with its underlying set of constraints $\mathcal{C}$. Finally, in order to model parameter passing, *diagonal elements* [8] are added to the primitive constraints: We assume that, for x, y ranging in *Var*, D contains the constraints d_{xy} which satisfy the following axioms:

(i) $\mathit{true} \vdash d_{xx}$, $\qquad$ (ii) if $z \neq x, y$ then $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$,
(iii) $x \neq y$ then $d_{xy} \sqcup \exists_x(c \sqcup d_{xy}) \vdash c$.

Note that if $\mathbf{C}$ models the equality theory, then the elements d_{xy} can be thought of as the formulas $x = y$. In the following $\exists_x(c)$ is denoted by $\exists_x c$ with the convention that, in case of ambiguity, the scope of $\exists_x$ is limited to the first constraint subexpression. (So, for instance, $\exists_x c \sqcup d$ stands for $\exists_x(c) \sqcup d$.)

The basic idea underlying ccp is that computation progresses via monotonic accumulation of information in a global store. Information is produced by the concurrent and asynchronous activity of several agents which can add (tell) a constraint to the store. More precisely, given a store d, the agent $\mathbf{tell}(c) \rightarrow A$ updates the store to $c \sqcup d$ and then behaves like the agent A. Dually, agents can also check (ask) whether a constraint is entailed by the store, thus allowing synchronization among different agents. So the action $\mathbf{ask}(c)$ represents a guard, i.e. a test on the current store d, whose execution does not modify d: if $d \vdash c$ then $\mathbf{ask}(c)$ is *enabled* (or satisfied) in d, otherwise $\mathbf{ask}(c)$ is suspended. Nondeterminism arises by introducing a *guarded choice* operator: The agent $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$ nondeterministically selects one $ask(c_i)$ which is enabled in the current store and then behaves like A_i. If no guard is enabled, then this agent *suspends*, waiting for other (parallel) agents to add information to the store. *Deterministic* ccp is obtained by imposing the restriction $n = 1$ in the above construct. The $\|$ operator allows one to express parallel composition of two agents $A\|B$ and it is usually described in terms of interleaving. Finally a notion of locality is obtained by introducing the agent $\exists x A$ which behaves like A, with x considered *local* to A.

When querying the store for some information which is not present (yet) a ccp agent will simply suspend until the required information has arrived. In real-time applications however often one cannot wait indefinitely for an event. Consider for example the case of a bank teller machine. Once a card is accepted and its identification number has been checked, the machine asks the authorization of the bank to release the requested money. If the authorization does not arrive within a reasonable amount of time, then the card should be given back to the customer. A real-time language should then allow us to specify that, in case a given time bound is exceeded (i.e. a time-out occurs), the wait is interrupted and an alternative action is taken. Moreover in some cases it is also necessary to abort an active process A and to start a process B when a specific event occurs (this is usually called *preemption* of A). For example, according to a typical pattern, A is the process controlling the normal activity of some physical device, the event indicates some abnormal situation and B is the exception handler.

In order to enrich ccp agents with such real-time mechanisms, we introduce a *discrete global clock* and assume that ask and tell actions take one time-unit. Computation evolves in steps of one time-unit, so called clock-cycles. We consider action prefixing as the syntactic marker which distinguishes a time instant from the next one. So $\mathbf{tell}(c) \rightarrow A$ has now to be regarded as the agent which updates the current store by adding c and then, at the *next* time instant, behaves like A. Analogously, if c is entailed by the current store then the agent $\mathbf{ask}(c) \rightarrow A$ behaves like A at the next time instant. If c is not entailed at time t then the agent is suspended, i.e. at time $t+1$ it is checked again whether the store entails c [1]. Note that if a $tell(c)$ action is performed at time t then the updated store will be visible only from time $t+1$ onwards, since a *tell* takes one time-unit to be completed. Thus, for example, the agent $A:\ (ask(c) \rightarrow stop) \parallel (tell(c) \rightarrow stop)$ evaluated in the empty store will take two time-units to successfully terminate.

Furthermore we make the assumption that parallel processes are executed on different processors, which implies that at each moment every enabled agent of the system is activated. This assumption gives rise to what is called *maximal parallelism* and, for example, implies that previous agent A evaluated in the store c terminates in one time-unit. The time in between two successive moments of the global clock intuitively corresponds to the response time of the underlying constraint system. Thus essentially in our model all parallel agents are synchronized by the response time of the underlying constraint system.

So far we have only described a timed interpretation of the usual ccp combinators. We still have to introduce the notions of *time out* and *preemption* which, as previously mentioned, are essential to any real-time language. Often *weak preemption* is sufficient, i.e. it is acceptable having a unit delay between the detection of the event and the consequent action. However, there are some time critical applications (see [14, 1]) in which *strong preemption* is required: The abort of a process and the execution of the new one must happen at the same time of the detection of the event. We will consider here a form of weak preemption: The abort of a process and the *start* of the new one happen at the same time of the detection of the event. However, the result of the execution of the new process will be visible only in the next time instant. As we discuss later, this choice allows us to obtain a programming paradigm useful for many applications, while maintaining a simple semantic model.

In general, as discussed in [13], the essence of the real time notions mentioned above is in the ability to detect the *absence* of an event, as well as its presence. Such a detection can interrupt a process and trigger some alternative actions. Since events in ccp can be expressed by the presence (more precisely, entailment) of a constraint in the store, we are lead to the following timing construct

$$\mathbf{now}\ c\ \mathbf{then}\ A\ \mathbf{else}\ B.$$

which is similar to the analogous construct in [13]. However, according to our notion of time interval, we interpret the above construct in terms of instantaneous reaction as follows: If c is entailed by the store at the current time instant then the above agent behaves as A at the current time instant, otherwise at the current time instant it behaves as B.

As we will show in Section 3, we can simulate the other typical real time constructs in terms of the **now then else** construct. Therefore we end up with the following syntax.

[1]The extension to the non-deterministic case is immediate.

Definition 2.2 [*tccp* Language] Assuming a given cylindric constraint system $\mathbb{C}$ the syntax of *agents* is given by the following grammar:

$$\begin{array}{rcl} A & ::= & \mathbf{stop} \mid \mathbf{tell}(c) \rightarrow A \mid \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i \mid \\ & & \mathbf{now}\ c\ \mathbf{then}\ A\ \mathbf{else}\ B \mid A \parallel B \mid \exists X A \mid p(X) \end{array}$$

where the c, c_i are supposed to be *finite constraints* (i.e. algebraic elements) in $\mathcal{C}$. A ccp *process* P is then an object of the form $D.A$, where D is a set of procedure declarations of the form $p(X) :: A$ and A is an agent.

2.1 Operational semantics

The operational model of *tccp* can be formally described by a standard transition system $T = (Conf, \longrightarrow)$ where we assume that each transition corresponds with one clock-cycle. Configurations (in) *Conf* are pairs consisting of a process and a constraint in $\mathcal{C}$ representing the common *store*. The transition relation $\longrightarrow \subseteq Conf \times Conf$ is the least relation satisfying the rules **R1-R8** in Table 1 and characterizes the (temporal) evolution of the system. So, $\langle A, c\rangle \longrightarrow \langle B, d\rangle$ means that if at time t we have the process A and the store c then at time $t+1$ we have the process B and the store d.

Let us now briefly discuss the rules in Table 1. The agent **stop** represents successful termination, so it cannot make any transition. Rule **R1** shows that we are considering here the so called "eventual" tell: The agent $\mathbf{tell}(c) \rightarrow A$ adds c to the store d without checking for consistency of $c \sqcup d$, and then behaves as A at the next time instant. Note that the updated store $c \sqcup d$ will be visible only starting from the next time instant since each transition step involves exactly one time-unit. According to rule **R2** the choice operator gives rise to global non-determinism: The external environment can affect the choice since $ask(c_j)$ is enabled at time t (and A_j is started at time $t+1$) iff the store d entails c_j, and d can be modified by other agents. The rules **R3** and **R4** show that the agent **now** c **then** A **else** B behaves as A or B depending on the fact that c is or is not entailed by the current store. Note that the evaluation of the guard is instantaneous: If $\langle A, d\rangle$ ($\langle B, d\rangle$) can make a transition at time t and c is (is not) entailed by the store d, then the agent **now** c **then** A **else** B can make the same transition at time t. Rules **R5** and **R6** model the parallel composition operator in terms of *maximal parallelism*: The agent $A \parallel B$ executes in one time-unit all the initial enabled actions of A and B. The agent $\exists X A$ behaves like A, with X considered *local* to A. To describe locality in rule **R7** the syntax has been extended by an agent $\exists^d X A$ where d is a local store of A containing information on X which is hidden in the external store. Initially the local store is empty, i.e. $\exists x A = \exists^{true} X A$. Rule **R8** treats the case of a procedure call when the actual parameter differs from the formal parameter: It identifies the formal parameter as a local alias of the actual parameter. For a call involving the formal parameter a simple body replacement suffices (rule **R9**) since we are dealing with a call by name parameter mechanism.

Using the transition system described by (the rules in) Table 1 we can now define our notion of observables. Here and in the sequel we assume a given fixed set of declarations D and we assume that P is a closed process (namely, every procedure occurring in P is declared in D). We denote by $\longrightarrow^*$ the reflexive and transitive closure of $\longrightarrow$.

Definition 2.3 Let P be a process. We define

$$\mathcal{O}(P) = \{\langle c, d\rangle \mid\ c \in \mathcal{C} \text{ and there exists } Q \text{ s.t. } \langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow\}.$$

R1	$\langle \mathbf{tell}(c) \rightarrow A, d\rangle \longrightarrow \langle A, c \sqcup d\rangle$	
R2	$\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, d\rangle \longrightarrow \langle A_j, d\rangle$	$j \in [1,n]$ *and* $d \vdash c_j$
R3	$\dfrac{\langle A, d\rangle \longrightarrow \langle A', d'\rangle}{\langle \mathbf{now}\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, d\rangle \longrightarrow \langle A', d'\rangle}$	$d \vdash c$
R4	$\dfrac{\langle B, d\rangle \longrightarrow \langle B', d'\rangle}{\langle \mathbf{now}\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, d\rangle \longrightarrow \langle B', d'\rangle}$	$d \not\vdash c$
R5	$\dfrac{\langle A, c\rangle \longrightarrow \langle A', c'\rangle \quad \langle B, c\rangle \longrightarrow \langle B', d'\rangle}{\langle A \parallel B, c\rangle \longrightarrow \langle A' \parallel B', c' \sqcup d'\rangle}$	
R6	$\dfrac{\langle A, c\rangle \longrightarrow \langle A', c'\rangle \quad \langle B, c\rangle \not\longrightarrow}{\begin{array}{c}\langle A \parallel B, c\rangle \longrightarrow \langle A' \parallel B, c'\rangle \\ \langle B \parallel A, c\rangle \longrightarrow \langle B \parallel A', c'\rangle\end{array}}$	
R7	$\dfrac{\langle A, d \sqcup \exists_x c\rangle \longrightarrow \langle B, d'\rangle}{\langle \exists^d X A, c\rangle \longrightarrow \langle \exists^{d'} X B, c \sqcup \exists_x d'\rangle}$	
R8	$\dfrac{\langle \exists^{d_{xy}} X A, c\rangle \longrightarrow \langle B, d\rangle}{\langle p(Y), c\rangle \longrightarrow \langle B, d\rangle}$	$p(X) : -A \in D,\ X \neq Y$
R9	$\dfrac{\langle A, c\rangle \longrightarrow \langle B, d\rangle}{\langle p(X), c\rangle \longrightarrow \langle B, d\rangle}$	$p(X) : -A \in D$

Table 1: The transition system for *tccp*.

So we observe the input/output behaviour of finite computations. Note that the above notion of observables abstracts from time. Alternatively, we could take into account the intermediate results of computations by considering sequences of constraints obtained from the relation $\longrightarrow$ in the obvious way. However, as we will show later, the resulting denotational model would be essentially the same (modulo a simple abstraction). For similar reasons as in the untimed case, the semantics which associates to a process P its observables $\mathcal{O}(P)$ is not compositional. We defer to section 4 the discussion of this point and the definition a simple compositional model.

Previous discussion shows that the standard *ccp* computational model can be extended very smoothly to incorporate a notion of time. A point which is worth mentioning here is that, differently from the untimed case, we cannot replace $\mathbf{tell}(c) \rightarrow A$ for $\mathbf{tell}(c)$ in the syntax of *tccp*. In fact, if the tell is eventual then in the untimed case $\mathbf{tell}(c) \rightarrow A$ can be equivalently rewritten as $\mathbf{tell}(c) \parallel A$. In the timed case previous two agents in general do not need to be equivalent. This is shown by the following.

Example 2.4 Consider the agents A : $(\mathbf{tell}(c) \rightarrow \mathbf{tell}(d)) \parallel B$ and A' : $(\mathbf{tell}(c) \parallel \mathbf{tell}(d)) \parallel B$ where B : $\mathbf{tell}(true) \rightarrow \mathbf{now}\ c \sqcup d\ \mathbf{then}\ \mathbf{tell}(ok)\ \mathbf{else}\ \mathbf{stop}$ and assume that

c, d, ok are different constraints such that $ok \vdash d \vdash c$. According to our operational model we have that $\langle true, ok\rangle \in \mathcal{O}(A') \setminus \mathcal{O}(A)$.

3 Some derived combinators

We show now how some typical real time programming idioms can be derived from the basic combinators of *tccp*.

Time out The timed guarded choice agent

$$\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i\ \mathbf{time\text{-}out}(m)\ B$$

waits at most m time units ($m \geq 0$) for the satisfaction of one of the guards. Before this time out the process behaves just like the guarded choice: As soon as there exist enabled guards, one of them and the corresponding branch is nondeterministically selected. After waiting for m time units, if no guard is enabled, the timed choice agent behaves as B. This agent can be defined inductively as follows. Let us denote by A the agent $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$. In the base case, $m = 0$, we define $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i\ \mathbf{time\text{-}out}(0)\ B$ as the agent

$$\begin{array}{l}\mathbf{now}\ c_1\ \mathbf{then}\ A\ \mathbf{else}\ (\mathbf{now}\ c_2\ \mathbf{then}\ A\ \mathbf{else} \\ \qquad\qquad \ldots\ (\mathbf{now}\ c_n \mathbf{then}\ A\ \mathbf{else}\ \mathbf{ask}(true) \rightarrow B)\ \ldots)\end{array}$$

For the inductive step we define $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i\ \mathbf{time\text{-}out}(m)\ B$ as

$$\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i\ \mathbf{time\text{-}out}(0) \left(\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i\ \mathbf{time\text{-}out}(m-1)\ B \right).$$

It is immediate to check that the above inductively defined agent has the expected operational behaviour. Consider for example the base case. If the current store entails one of the guards c_i we have that by rule **R3** the agent $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$ is executed immediately, that is, in the next time instant one of the agents A_i (for which the corresponding guard c_i is enabled) is executed. Otherwise, the agent B is executed at the next time instant.

Watchdogs These are typical preemption primitives of such languages as ESTEREL. Watchdogs are used to interrupt the activity of a process on signal from a specific event: In our framework, since events are expressed by constraints, a watchdog can be defined as the process

$$\mathbf{do}\ A\ \mathbf{watching}\ c$$

which behaves as A, as long as c is not entailed by the store; when c is entailed, the process A is immediately aborted. Notice that, as discussed above, we have instantaneous reaction in the sense that A is aborted at the *same* time instant of the detection of the entailment of c. However, according to the computational model, if c is detected at time t then c has to be produced at time t' with $t' < t$. Thus we have a form of weak preemption.

Previous watchdog agent can be defined by induction on the structure of A as follows. In the following we use $\mathbf{now}\ c$ as a shorthand for $\mathbf{now}\ c\ \mathbf{then}\ \mathbf{stop}$.

$$\begin{array}{lll}
\textbf{stop} & \Rightarrow & \textbf{stop}, \\
\textbf{tell}(d) \rightarrow B & \Rightarrow & \textbf{now}\ d\ \textbf{else tell}(c) \rightarrow\ \textbf{do}\ B\ \textbf{watching}\ c, \\
\sum_{i=1}^{n} \textbf{ask}(c_i) \rightarrow A_i & \Rightarrow & \textbf{now}\ c\ \textbf{else} \sum_{i=1}^{n} \textbf{ask}(c_i) \rightarrow \textbf{do}\ A_i\ \textbf{watching}\ c, \\
\textbf{now}\ d\ \textbf{then}\ A\ \textbf{else}\ B & \Rightarrow & \textbf{now}\ c\ \textbf{else now}\ d\ \textbf{then do}\ A\ \textbf{watching}\ c\ \textbf{else} \\
 & & \quad \textbf{do}\ B\ \textbf{watching}\ c, \\
A \parallel B & \Rightarrow & \textbf{now}\ c\ \textbf{else do}\ A\ \textbf{watching}\ c \parallel \textbf{do}\ B\ \textbf{watching}\ c, \\
\exists X A & \Rightarrow & \textbf{now}\ c\ \textbf{else}\ \exists X\ \textbf{do}\ A\ \textbf{watching}\ c,
\end{array}$$

(So the agent on the rhs of the arrow $\rightarrow$ is the translation of the agent **do** A **watching** c, where A is the agent on lhs.) Analogously we can define the agent **do** A **watching** c **else** B which behaves as the previous watchdog and also activates the process B when A is aborted (i.e. when c is entailed).

4 Denotational semantics

In this section we give a denotational semantics for *tccp* programs. Denotationally we represent a (timed) computation by a sequence of the form $\langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle$, a so called *timed reactive sequence.* A pair $\langle c_i, d_i\rangle$ indicates that at time i the process itself produces c_i while at the same time its environment produces d_i. The set of all timed reactive sequences we denote by $\mathcal{S}$. Elements of $\mathcal{S}$ are denoted by $s, \ldots$. We define $D(A)s \subseteq \mathcal{S}$, i.e. the set of timed reactive sequences of A starting from the initial sequence s. Given the initial sequence s, the agent **Stop** does not modify it. Thus $D(\textbf{Stop})s = s$. The meaning of $\textbf{tell}(c) \rightarrow A$ is defined by

$$D(\textbf{tell}(c) \rightarrow A)s = \{s' \mid \text{ for some } d,\ s' \in D(A)(s \cdot \langle c, d\rangle)\}$$

where $s \cdot \langle c, d\rangle$ denotes the sequence resulting from appending $\langle c, d\rangle$ to s. Thus, given the initial sequence s, the agent A in $\textbf{tell}(c) \rightarrow A$ starts its computation, after execution of $\textbf{tell}(c)$, in the sequence $s \cdot \langle c, d\rangle$, where d represents the contributions of the environment, which occur at the same time as the execution of $\textbf{tell}(c)$. The meaning of $\Sigma_i \textbf{ask}(c_i) \rightarrow A_i$ is defined by

$$D(\Sigma_{i=1}^{n} \textbf{ask}(c_i) \rightarrow A_i)s = \ \bigcup_i \{s' \mid \begin{array}{l} \text{there exists } s'' \text{ and } d \text{ such that } s \rightarrow^* s'', \\ s'' \vdash c_i \text{ and } s' \in D(A_i)(s'' \cdot \langle \mathit{true}, d\rangle)\} \end{array}$$

where $s \vdash c$ holds if the least upper bound (lub for short) of all the constraints occurring in s entails c, and $\rightarrow^*$ denotes the reflexive transitive closure of the relation $\rightarrow$ between elements of $\mathcal{S}$ defined by:

$$s \rightarrow s' \text{ if } s \not\vdash c_j, \text{ for } 1 \leq j \leq n, \text{ and } s' = s \cdot \langle \mathit{true}, d\rangle, \text{ for some } d.$$

Given the initial sequence s, a sequence s' such that $s \rightarrow^* s'$ represents an extension of s which consists of a period of waiting for one of the constraints c_i. Note that during this waiting period only the environment is active. The addition of a pair $\langle \mathit{true}, d\rangle$ to the waiting period corresponds to the assumption that the ask takes one time-unit. The agent A_i then starts its computation in an extension of the initial computation s which consists of a waiting period after which c_i, for some $1 \leq i \leq n$, has arrived. The meaning of **now** c **then** A **else** B can be simply defined by:

$$D(\textbf{now}\ c\ \textbf{then}\ A\ \textbf{else}\ B)s = \{s' \mid s \vdash c \text{ and } s' \in D(A)s\} \cup \{s' \mid s \not\vdash c \text{ and } s' \in D(B)s\}$$

To describe denotationally the parallel composition we introduce the following (commutative) partial operator $\parallel \in \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$: Let $n \leq m$, and $d_i \vdash c_i \sqcup e_i$, for $1 \leq i \leq n$, then

$$\begin{array}{l} \langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle \parallel \langle e_1, d_1\rangle \cdots \langle e_m, d_m\rangle = \\ \langle c_1 \sqcup e_1, d_1\rangle \cdots \langle c_n \sqcup e_n, d_n\rangle \cdot \langle e_{n+1}, d_{n+1}\rangle \cdots \langle e_m, d_m\rangle \end{array}$$

In all other cases the parallel composition is undefined. Note that we require the two arguments of the parallel operator to agree at each point of time with respect to the environment. The condition $d_i \vdash c_i \sqcup e_i$ corresponds with that the environment of one component has to include the contributions of the parallel component. Now we can simply define

$$D(A \parallel B)s = D(A)s \parallel D(B)s$$

where $\parallel$ denotes the obvious extension of the above defined operator to set of sequences. To describe denotationally the hiding of local variables we introduce the operators $\exists^1_x, \exists^2_x \in \mathcal{S} \rightarrow \mathcal{S}$:

$$\exists^1_x(\langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle) = \langle \exists_x c_1, d_1\rangle \cdots \langle \exists_x c_n, d_n\rangle$$

and

$$\exists^2_x(\langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle) = \langle c_1, \exists_x d_1\rangle \cdots \langle c_n, \exists_x d_n\rangle$$

The operator $\exists^1_x$ thus removes at each point of time the information on the (local) variable x, that is, the information on x produced by the process itself. On the other hand, the operator $\exists^2_x$ removes at each point of time the information on the (global) variable x, that is, the information on x produced by the environment. Then we define

$$D(\exists X A)s = \{\exists^1_x s' \mid s' \in D(A)\exists^2_x s\}$$

Note that the operator $\exists^2_x$ removes information on the global x while $\exists^1_x$ removes the information on the local x. Recursion finally is defined as follows: Let $p(X)$ be declared as A. Then

$$D(p(X))s = D(A)s \ \text{ and } \ D(p(Y))s = D(\exists^{d_{xy}} X A)$$

in case the actual parameter Y differs from the formal parameter X. This recursive definition can be easily justified by a least fixed-point construction defined in terms of the cpo $\mathcal{P}(\mathcal{S})$ with the ordering of simple set-inclusion.

Correctness of the denotational semantics D with respect to the operational semantics is expressed by the following theorem:

Theorem 4.1 (Correctness) *For any agent* A,

$$O(A)c = \{d \mid \text{ there exists } \langle c, c\rangle \cdots \langle d, d\rangle \in D(A)\langle c, c\rangle\}.$$

Note that a sequence $\langle c_1, c_1\rangle \cdots \langle c_n, c_n\rangle$ represents a computation where the assumed contributions of the environment are already produced by the agent itself. It is straightforward to prove that such a sequence indeed corresponds with a computation as defined by the operational semantics. So the model defined by D is correct.

However this model introduces unnecessary distinctions. For example, considering the agents $A : \mathit{tell}(c \sqcup d) \rightarrow \mathit{tell}(c) \rightarrow \mathit{stop}$ and $B : \mathit{tell}(c \sqcup d) \rightarrow \mathit{tell}(d) \rightarrow \mathit{stop}$, we have that $D(A)\epsilon \neq D(B)\epsilon$ (ϵ denotes the empty sequence) while for any context $C[\,]$ and initial store

c we have that $O(C[A])c = O(C[B])c$. (A context $C[\,]$ is simply an agent with a 'hole', the agent $C[A]$ then represents the result of replacing the hole in C by A.) The point here is that once the stronger constraint $c \sqcup d$ has been produced, it does not matter whether c or d are produced. In order to identify agents like the previous ones, we introduce the following abstraction *inc* on sequences. This operation adds to each left component of a sequence s all the previous left components of s, thus transforming s into a left-increasing sequence. For $s = \langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle \in \mathcal{S}$ we then define

$$inc(s) = \langle c_1, d_1\rangle \cdot \langle c_1 \sqcup c_2, d_2\rangle \cdots \langle c_1 \sqcup c_2 \sqcup \ldots \sqcup c_n, d_n\rangle.$$

Moreover, we should not distinguish (left-increasing) sequences which differ only for the number of repetitions of the last element: For example, the agents C : *tell*$(c) \rightarrow$ *stop* and C' : *tell*$(c) \rightarrow$ *tell*$(c) \rightarrow$ *stop* should be identified[2]. Thus we need also the following abstraction *rep* on left increasing sequences. For $s \in \mathcal{S}$ we denote by left(s) the lub of the left parts of all the pairs in s. We define *rep* inductively as follows: $rep(\epsilon) = \epsilon$ and

$$rep(s \cdot \langle c, d\rangle) = \begin{cases} rep(s) & \text{if } left(s) \vdash c \\ s \cdot \langle c, d\rangle & \text{if } left(s) \not\vdash c \end{cases}$$

Let us define $D^{\alpha}(A)s$ as $rep(inc(D(A)s))$, where we denote by *rep* and *inc* the obvious extension of the above operators to sets of sequences. It is immediate to show that the model defined by D^{α} is also correct and compositional. Moreover we have that D^{α} does not introduce unnecessary distinctions, i.e. the semantics defined by D^{α} is fully abstract:

Theorem 4.2 *For any agents A and B, if $D^{\alpha}(A)s \neq D^{\alpha}(B)s$ for some s, then there exists a context C such that $O(C[A])c \neq O(C[B])c$, for some c.*

5 A comparison with (default) tcc and future research

A timed version of ccp, called `tcc`, and a further extension called `default tcc` have recently been introduced in [13] and [14]. To compare these approaches with our proposal let us first sketch briefly `tcc`. As in *tccp* and differently from the case of synchronous languages, computation in `tcc` takes a bounded period of time rather than being instantaneous. However, differently from our case, a time interval for `tcc` is identified with the time needed for a ccp process to terminate a computation. Computation evolves asynchronously in cycles: At each time interval a ccp deterministic process is executed. The process accumulates monotonically information in the store, according to the standard ccp computational model, until it reaches a "resting point", i.e. a terminal state in which no more information can be generated. The resting point is then seen as the marker which distinguishes time intervals. When the resting point is reached the absence of events can be checked and it can trigger actions in the next time interval. More precisely, the process A : `now` c `else` B is evaluated at "resting time": If the store obtained at the end of previous time interval does not entail c then A behaves as the process B in the next time interval, otherwise A is discarded. A *unit delay* primitive is also present: `next B` is the process which behaves like B in the next time interval.

[2]On the other hand, assuming that c, d, ok are different constraints such that $ok \vdash d \vdash c$, the agents B : *tell*$(c) \rightarrow$ *tell*$(d) \rightarrow$ *stop* and B' : *tell*$(c) \rightarrow$ *tell*$(c) \rightarrow$ *tell*$(d) \rightarrow$ *stop* must be distinguished. In fact, for A : *tell*$(true) \rightarrow$ (*now d then tell*(ok) *else stop*), we have that $\langle true, ok\rangle \in \mathcal{O}(B \parallel A) \setminus \mathcal{O}(B' \parallel A)$.

A crucial design decision for `ttc` was to enforce the programmer to transfer explicitly the information from one time interval to the next one. At the end of a time interval all the constraints accumulated are discarded, as well as all the processes which are not argument to a `next` or to a (satisfied) `now else` command. Thus basically a `tcc` program specifies for each moment in time a ordinary (deterministic) ccp program to be executed at that particular moment. Since the next moment in time occurs when the ccp program has reached a resting point, to ensure that the next time instant is reached such an ordinary ccp program has to be a finite agent.

Our starting point however is to interpret action-prefixing itself as the next-time operator. In our framework a time interval is identified with the time the underlying constraint system needs to respond to the initial actions of all the agents of the system, that is, to accumulate all the told constraints and to answer all the ask's. Thus a real-time program in our case is basically just a usual ccp program (apart from the `now` construct). The real-time aspects are mainly implicit in the interpretation of the basic actions and the interpretation of action-prefixing. As such the style of programming in *tccp* is more similar to the usual one for asynchronous monotonic languages, also because in our framework the global store persists from one moment to the other. For example, the operators of hiding local variables and recursion do not differ in an essential manner from their 'untimed' versions. This is to be contrasted with the language `tcc`, where the fact that the store is killed each next moment complicates the real-time interpretation of recursion and hiding of local variables (the information on the local variables is killed too each next time instant). Also we do not need any syntactic restriction to ensure that the next time instant is reached, since at each moment there are only a finite number of parallel agents and the next moment in time occurs as soon as the underlying constraint system has responded to the initial actions of all the current agents of the system.

`Default tcc` is essentially like `tcc`, except that at each time interval a default ccp program (rather then a ccp program) is executed. Thus previous discussion applies also to the case of `Default tcc`. The advantage of `Default tcc` over `tcc` and *tccp* is that the former language allows one to express strong preemption by using agents of the form $c \leadsto A$: If c is not entailed by the current store then A is immediately evaluated. Differently from the case of *tccp*, the result of this evaluation is visible within the same time interval. This increased expressive power of `Default tcc` comes with a price since, as previously mentioned, in general strong preemption can cause paradoxes: If the agent A produces c then the construct $c \leadsto A$ could have ambiguous interpretations. To avoid these problems `Default tcc` uses *assumptions* about the *future* evolution of the system. If c is absent when evaluating $c \leadsto A$, then it is *assumed* that c will also be absent in the future, i.e. A and the other processes being evaluated in parallel cannot produce c. These assumptions semantically are modeled by using pairs of constraints. The pair (c, d) in the denotation of an agent A means that A reaches a resting point c given the guess d about the final result. The resulting model however, as discussed in [14], does not allow for the definition of first-order existentials. Thus, differently from tccp, `Default tcc` does not allow hiding.

Moreover, the simplicity of both the `tcc` and the `Default tcc` models, sequences of constraints and sequences of pair of constraints, is due to the restriction to deterministic programs. An extension to non-determinism would require complicated models based on sequences of sequences. On the other hand, our real-time extension allows a simple operational and denotational fully abstract semantics for non-deterministic real-time programs.

Concluding, we have defined an extension of ccp to model real-time which, altough inspired by the motivations in [13], as discussed above differs from `tcc` and `default tcc` both in the language design and in the semantic model. We believe that our proposal provides a smooth extension of ccp and therefore allows to retain as much as possible the usual ccp programming style also for real time applications. Moreover, the simplicity of our semantic model seems a promising basis to define tools for the verification and the analysis of *tccp* programs, following the guidelines of [3] and [5]. In particular, we are now studying an extension based on temporal logic [10] of the proof system defined in [3] to reason about the correctness of *tccp* programs.

References

[1] G. Berry. Preemption in concurrent systems. In *Proc of FSTTCS*, volume 761 of LNCS, pages 72–92. Springer-Verlag, 1993.

[2] G. Berry and G. Gonthier. The ESTEREL programming language: Design, semantics and implementation. *Science of Computer Programming*, 19(2):87-152, 1992.

[3] F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving Concurrent Constraint Programs Correct. In *Proc. of Twentyfirst POPL*, ACM Press, 1994.

[4] F.S. de Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of TAPSOFT/CAAP, LNCS 493*, pages 296–319. Springer-Verlag, 1991.

[5] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Compositional Analysis for Concurrent Constraint Programming. In *Proc. of Eighth LICS*, pages 210–221. IEEE Computer Society Press, 1993.

[6] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language LUSTRE. In *Special issue on Another Look at Real-time Systems*, Proceedings of the IEEE, 1991.

[7] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, pages 231-274, 1987.

[8] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras (Part I)*. North-Holland, 1971.

[9] P. Le Guernic, M. Le Borgue, T. Gauthier, and C. Le Marie. Programming real time applications with SIGNAL. In *Special issue on Another Look at Real-time Systems*, Proceedings of the IEEE, 1991.

[10] Z. Manna and A. Pnueli. The temporal logic of reactive systems. *Springer-Verlag, 1991.*

[11] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989. Published by The MIT Press, U.S.A., 1991.

[12] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of POPL*, pages 232–245, 1990.

[13] V.A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of Timed Concurrent Constraint Programming. In *Proc. of LICS*, 1994.

[14] V.A. Saraswat, R. Jagadeesan, and V. Gupta. Default Timed Concurrent Constraint Programming. In *Proc. of POPL*, 1995.

[15] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of Concurrent Constraint Programming. In *Proc. of POPL*, 1991.

[16] D. Scott. Domains for denotational semantics. In *Proc. of ICALP*, 1982.

Extending CAML Light to perform distributed computation[*]

J. L. Freire Nistal
LFCIA, University of A Coruña, 15071 La Coruña, Spain
`freire@dc.fi.udc.es` Phone: +81-102552, Fax: +81-102736

B. B. Fraguela Rodríguez
Dpto. de Electrónica y Sistemas, University of A Coruña
`basilio@des.fi.udc.es`

V. M. Gulías Fernández
Dpt. of Computer Science, University of Yale
`gulias@cs.yale.edu`

Abstract

In this paper an extension of Caml Light language that attempts to achieve distributed computation using a client-server model on a set of computers in a network is presented. To succeed in doing it, we have used a distributed-memory multiprocessor machine simulator in which we have implemented a system that can be used to construct distributed programs maintaining some important characteristics of functional programs such as referential transparency and the determinism of Caml Light evaluation order. The modifications required to transform a given sequential program in such a way that it exploits the parallelism derived from our implementation are minimal.

Keywords: Distributed and parallel architectures, data types and data structures, functional programming, program transformation

[*]Supported by Xunta de Galicia XUGA10502B94

1 Introduction and Previous Work

This work is aimed to use the facilities of the functional languages to exploit the *parallelism*. In sequential Von Neumann machines (*control flow* machines), instructions are executed sequentially, controlled by a program counter. A new philosophy has been proposed to improve the performance: the *data flow* machines. These allow us to forget about counters and instructions flow because instructions are carried out as data and operands are available, leaning on the properties of the functional paradigm. To illustrate this possibility with a real implementation, we have used the Caml Light language [1, 2, 3], a variant of CAML. Further information about this language can be seen in [4].

We have employed the PVM[1] software package [5] which allows us to write programs that exploit the distributed computation model on heterogeneus networks with the unique imposition that all the computers work under the Unix operating system (also under OSF-1 for Alpha machines).

This paper appears as an evolution of [6, 7], in which a heterogeneous computation model is presented in order to integrate functional and imperative modules on a client-server architecture. In [8] the same model refined is discussed and applied to an example, and a mechanism for the automatic (or semiautomatic) construction of the Functional Lenguage and Operate System interface based on the type of the kernel functions we want to use from the client side is introduced.

2 Breaking down the synchronism of the typical evaluation sequence

Most of the functional languages (or extensions of them) that pursue parallel computation, employ the *non-strict* (*lazy*) evaluation order since there is not synchronism between the demand of a computation and the obtention of the result. In this way, it is possible to introduce *annotations* in the code to suggest the moment in which the evaluation of an expression should start, the temporal dependences of the evaluation in relation to evaluations of other expressions, and the end of the evaluation, like the para-language of Haskell

[1]Other possible systems available are ISIS, P4, Express and Linda

presented by P.Hudak in [9].

The synchronism introduced in the *strict* languages prevents from adopting this solution (at least immediately). Some of these languages incorporate primitives to generate new processes or threads, controlled all the time by the programmer, and to handle the interprocess communication by side-effects in a similar way to I/O handling, thus losing referential transparency.

To solve this problem we adopted a solution resembling that used in lazy languages, but preserving the Caml Light evaluation mechanism (strict evaluation) and its referential transparency. The idea consists of making the request to a server in such a way that we do not need to wait for the remote result, continuing the evaluation of other expressions, carrying them in parallel. If we should wait for the result of the remote evaluation, the parallelism would not be possible. The solution resides in the construction of a new datatype, whose behaviour is quite similar to a lazy datatype (a deferred datatype) `Remoteval`.

A client-server model has been used so that remote evaluation requests can be performed by the Caml Light function `request` which has as its arguments an integer identifying one suitable service, and the arguments to this service with type `'a`, and that returns a value of type `'b Remoteval` where `'b` is the type of the value returned by the service (`int -> 'a -> 'b Remoteval`). If the function has more than one argument, say n of types `'a1`, `'a2`,..., `'a`n, we can transform them into only one parameter using the product type (uncurry): `'a = 'a1 * 'a2 * ... * 'a`n. Since `'a` can be any type, we can define and use services of multiple arguments by joining them together in a n-tuple that will be unfolded in the server.

On the other hand, since `request` achieves the evaluation request to the server, the value returned by the function, an instance of `Remoteval`, does not contain initially the result of the evaluation. `Remoteval` is an ADT (Abstract Data Type) which can be used to encapsulate the access to the result of the remote evaluation (function `val`) when needed. This ADT (defined in a Caml Light module) hides an imperative structure, safeguarding the determinism in the evaluation and the referential transparency due to its access function `val`.

The ADT is created at request time, storing the result when it arrives. If we try to access the data contained in it before they arrive, the process must wait until they are available. There are two different approaches about value reception: