PROCEEDINGS

GULP-PRODE
1995

Editors

MARIA
ALPUENTE

MARIA I.
SESSA

PROCEEDINGS

Editors
MARIA ALPUENTE
MARIA I. SESSA

11-14 September 1995
MARINA DI VIETRI - ITALY

Maria I. Sessa María Alpuente Frasnedo (Eds.)

GULP-PRODE'95

JOINT CONFERENCE ON DECLARATIVE PROGRAMMING

Marina di Vietri sul Mare, Italy

September 11-14 1995

PROCEEDINGS

UNIVERSITA' DEGLI STUDI
DI SALERNO

Printed with the support of:

University of Salerno, Italy

Dip. Informatica e Applicazioni "R.M. Capocelli"
University of Salerno, Italy

Printed in Italy on July 1995 by:
POLIGRAF Press
v. Pio XI 110
84125 Salerno, Italy.

Typesetting:
Camera ready by authors/editors

Cover:
Pottery art of Vietri sul Mare
(private collection)

Preface

The *Second Joint Conference on Declarative Programming* GULP-PRODE'95 joins together the 10th Italian GULP Conference on Logic Programming and the 5th Spanish PRODE Congress on Declarative Programming.

GULP-PRODE'95 was held in Marina di Vietri (Italy) at the IIASS institute on September 11-14, 1995, following the eight previous GULP conferences in Genova (1986), Torino (1987), Roma (1988), Bologna (1989), Padova (1990), Pisa (1991), Tremezzo (1992) and Gizzeria (1993), the three previous PRODE meetings in Torremolinos (1991), Madrid (1992) and Blanes (1993), and the *First Joint Conference on Declarative Programming* GULP-PRODE'94 in Peñíscola, Spain.
GULP-PRODE'95 has been organized by the Universitá di Salerno.

The technical program for the Conference included 43 full communications, 6 short communications, 4 guest lectures and 2 guest talks. The papers in this book are printed in their order of presentation at the Conference, with communications grouped into thematic sessions. The papers were selected from 56 received submissions. All the papers were evaluated by at least two reviewers. The Program Committee met at the Universitá di Salerno to select the 49 papers which are included in this volume as full or short communications.

In addition to the contributed papers, GULP-PRODE'95 featured four outstanding lectures by: *Krzysztof R. Apt* (CWI Amsterdam), *Patrick Cousot* (Ec. Nor. Sup. Paris), *Robert A. Kowalski* (Imp. Col. London), and *Giorgio Levi* (Univ. Pisa). The lecture by K. Apt was presented by *Elena Marchiori.* Two distinguished talks were also given by *Dale Miller* (Pennsylvania Univ.) and *Luis M. Pereira* (Univ. Nova de Lisboa).

We express our gratitude to all members of the Program Committee and to all outside referees who offered their expertise in the review process. We extend our sincere thanks to all authors who submitted papers and to all conference participants.
We would also like to thank all institutions supporting the Conference, with special thanks to the Universitá di Salerno and to the Dipartimento di Informatica e Applicazioni "R.M. Capocelli". Moreover, we gratefully aknowledge the broad of directors of the IIASS and the secretary of this institute, Ornella De Pasquale, for her efficient support.

Finally, we wish to especially highlight the contribution of the Organizing Committee, whose work made the Conference possible, and give special thanks to Andrea F. Abate, Bruno Carpentieri, Filomena Ferrucci, Vincenzo Loia, Alfonso Sessa and Giuliana Vitiello. We also wish to express our deep gratefulness to the Organizing Committee of GULP-PRODE'94 for their previous work and experience which have been extremely useful.

María Alpuente Frasnedo
Maria I. Sessa
Editors
Salerno, June 1995

GULP-PRODE'95
Joint Conference on Declarative Programming

Program Chair

Maria I. Sessa	U. Salerno

Program co-Chair

María Alpuente Frasnedo	U.P. Valencia

Program Committee

Annalisa Bossi	U. Calabria	Ernesto Burattini	CNR Napoli
Luigia Carlucci Aiello	U. Roma	M. Celma Giménez	U.P. Valencia
Paolo Ciancarini	U. Bologna	Nicoletta Cocco	U. Venezia
Stefania Costantini	U. Milano	Veronica Dahl	S.F.U. Canada
Moreno Falaschi	U. Udine	Gilberto Filé	U. Padova
Pere García Calves	IIIA Blanes	María J. García de la Banda	U.P. Madrid
José C. González Cristobal	U.P. Madrid	María T. Hortalá González	U.C. Madrid
Giorgio Levi	U. Pisa	Paqui Lucio Carrasco	U. País Vasco
Alberto Martelli	U. Torino	Maurizio Martelli	U. Genova
Juan J. Moreno Navarro	U.P. Madrid	Robert Nieuwenhuis	U.P. Catalunya
Eugenio Omodeo	U. Salerno	Giuliano Pacini	U. Venezia
Catuscia Palamidessi	U. Genova	Dino Pedreschi	U. Pisa
Inmaculada Pérez de Guzmán	U. Málaga	Alberto Pettorossi	U. Roma
Ernesto Pimentel Sánchez	U. Málaga	María J. Ramírez Quintana	U. P. Valencia
Mario Rodríguez Artalejo	U. C. Madrid	José J. Ruz Ortiz	U. C. Madrid
Domenico Saccá	U. Calabria	Roberto Serra	F.F. Ravenna
Genny Tortora	U. Salerno	Franco Turini	U. Pisa
Felisa Verdejo	UNED		

Organizing Committee
Andrea Abate
Bruno Carpentieri
Filomena Ferrucci
Vincenzo Loia
Giuliana Vitiello

Technical Support
Alfonso Sessa

Referees

A. F. Abate
L. Araujo
F. Arcelli
A. Asperti
R. Barbuti
C. Böhm
P. Bonatti
A. Bottoni
C.G. Brown
M. Bugliesi
N. Cancedda
A. Casanova
S. Castellani
S. Contiero
M. Coppo
A. del Pozo Prieto
S. Etalle
M. Fabris
F. Ferrucci
F. Formato
M. Gabbrielli
J. Garcia-Martin
M. Gaspari
S. Greco
G. Guerrini
A. Guglielmi
A. Herranz-Nieva
C. Laneve
G.A. Lanzarone
J. Levy
V. Loia
F.J. Lopez-Fraguas
S. Lucas
P. Mancarella
G. Manco
G. Mascari
M.C. Meo
A. Messora
M. Napoli
M. Navarro
N. Olivetti
L. Palopoli
R. Peña
M. Proietti
J. Puyol-Gruart
A. Raffaetá
F. Ranzato
C. Renso
R.O. Rodriguez
S. Rossi
A. Rubio
S. Ruggieri
P. Rullo
A. Sanchez Ortega
F. Scozzari
U.P. Vasco
G. Vidal
J. O. Villaroja
G. Vitiello
E. Zaffanella

Foreword

The Gruppo Ricercatori ed Utenti di Logic Programming (GULP, which stands for Logic Programming Researchers and Users Group), is an affiliate to the Association of Logic Programming (ALP). The goals of the group are to make Logic Programming more popular and to create opportunities for the exchange of experiences and information between researchers and users working in the field for both public and private organizations.
To this purpose GULP promotes many different activities such as the exchange of information among its members and the organization of workshops, advanced schools and its annual Conference.

Starting from 1994 our annual Conference is held jointly with the Spanish Conference PRODE on Declarative Programming. This has represented a significant step towards the exchange of research experience among European Latin countries.
The main aims of the Conference are:
1) to serve as an occasion for those working in this area which are interested in meeting and exchanging experiences;
2) to illustrate the current state of the art in the area through invited talks given by well known researchers;
3) to enable students and researchers to learn more about logic and declarative programming by means of introductory tutorials.

This year we will celebrate a special anniversary, namely the first ten years of GULP and the tenth Conference. The previous annual conferences were held in Genoa, Turin, Rome, Bologna, Padua, Pisa, Tremezzo, Gizzeria Lido, and, as a GULP-PRODE Conference, in Peñíscola (Spain).
The scientific program of this joint Conference GULP-PRODE includes papers from colleagues from several European countries. The large international participation, considered together with the good technical quality of the papers accepted for presentation, is a further confirmation of the success of this joint event.

On behalf of GULP I would like to thank Maria Sessa and all the other colleagues of the Universitá di Salerno for the organization of this year Conference.

Maurizio Martelli
President of the GULP

Contents

INVITED LECTURES

Arrays, Bounded Quantification and Iteration in Logic and Constraint Logic Programming 19
K.R. Apt

Completeness in Abstract Interpretation 37
P. Cousot

Logical Foundations for Multi-agent Systems 39
R. Kowalski

On the Abstract Diagnosis of Logic Programs 41
G. Levi, M. Comini, and G. Vitiello

GUEST TALKS

Observations about Using Logic as a Specification Language 61
D. Miller

Parallel Logic Programming with Extensions 71
L.M. Pereira, J.C. Cunha, and L. Damas

CONCURRENCY

Domain Independent Ask Approximation in CCP 89
E. Zaffanella

Modeling Real-Time in Concurrent Constraint Programming 101
F.S. de Boer, and M. Gabbrielli

Extending CAML Light to Perform Distributed Computation 113
J.L. Freire Nistal, B.B. Fraguela Rodríguez, and V.M. Gulías Fernández

A Logic Language Based on GAMMA-Like Multiset Rewriting 125
P. Ciancarini, D. Fogli, and M. Gaspari

DEDUCTIVE DATABASES

Side Effect Analysis for Logic-Based Planning 139
K. Eshghi, and M. Mowbray

Downward Refinement of Hierarchical Datalog Theories 148
F. Esposito, N. Fanizzi, D. Malerba, and G. Semeraro

Integrity Constraints Evolution in Deductive Databases 160
D. Montesi, and F. Turini

Declarative Reconstruction of Updates in Logic Databases:
a Compilative Approach 169
M. Carboni, V. Foddai, F. Giannotti, and D. Pedreschi

NEGATION

An Introduction to Regular Search Spaces 183
A. Momigliano, and M. Ornaghi

A Framework for a Transformational Approach to Negation 195
J. Humet

Ordered Logic and its Relationships to other Logic Programming Formalisms 207
F. Buccafurri

Analysis of SLDNF for Local CLP 219
A. Bottoni

A Semantics for the Kakas-Mancarella Procedure for Abductive
Logic Programming 231
F. Toni

HIGHER-ORDER

Implementing Higher-Order Term-Rewriting for Program Transformation
in λProlog 245
F. Arcelli, and F. Formato

The Undefined Function Differs from the Pointwise Undefined Function 257
W. Dosch

EXTENSIONS AND INTEGRATION

LOO: An Object Oriented Logic Programming Language 271
P. Mancarella, A. Raffaetá, and F. Turini

Forum & Objects 283
G. Delzanno, and M. Martelli

Three-Valued Semantics for Extended Logic Programs 295
P.A. Bonatti, and L. Giordano

Constructing Logic Programs with Higher-Order Predicates 307
J.F. Nilsson, and A. Hamfelt

Petri Nets and Linear Logic: a Case Study for Logic Programming 313
I. Cervesato

APPLICATIONS

GRAMPAL: A Morphological Processor for Spanish Implemented in Prolog 321
A. Moreno, and J.M. Goñi

A Declarative Approach to the Design and Realization of Graphic Interfaces 332
D. Aquilino, D. Apuzzo, and P. Asirelli

IMPLEMENTATIONS

Improving the Efficiency of Dynamic Modular Logic Languages 347
A. Ciampolini, E. Lamma, and P. Mello

Lazy Narrowing on an Abstract Machine by Means of Examples 359
E. Ullán-Hernández

Exploiting Expression- and Or-Parallelism for a Functional
Logic Language 371
W. Hans, St. Winkler, and F. Sáenz

An Effective Algorithm for Compiling Pattern Matching Keeping
Laziness 383
P. Palao, and M. Núñez

A Prolog Implementation of Kem 395
A. Artosi, P. Cattabriga, and G. Governatori

Explicit Implementation of a Constraint Solving Mechanism in
a Relational Programming System 401
P. Bellot, O. Camp, and C. Matiachoff

LINEAR LOGIC

A Linear Logic Programming Language with Parallel and Sequential Conjunction — 409
P. Bruscoli, and A. Guglielmi

A Structural (Meta-Logical) Semantics for Linear Objects — 421
G. Manco, and F. Turini

THEORY AND FOUNDATIONS

Computational Properties in Context-Sensitive Rewriting — 435
S. Lucas

Minimal Set Unification — 447
P. Arenas-Sánchez, and A. Dovier

A Model Tree Computation of the Strong Well-Founded Semantics — 459
C. Papp

Mónadas para la Comunicación de Objetos Funcionales — 471
J.E. Gallardo, P. Guerrero, and B.C. Ruiz

La Potencia Expresiva de los Catamorfismos — 477
C. Gregorio, M. Núñez, and P. Palao

TRANSFORMATION AND SYNTHESIS

A Tabulation Transformation Tactic Using Haskell Arrays — 485
C. Pareja-Flores, R. Peña, and J.A. Velázquez-Iturbide

Síntesis de Programas Lógicos: Marco Constructivo — 497
F.J. Galán Morillo, and M. Toro Bonilla

SEMANTICS

A Transitional Semantics of Full Prolog — 511
P. Degano, and C. Priami

Analysis of Pure PROLOG Programs — 521
G. Levi, and D. Micciancio

Compositionality in SLD-derivations and their Abstractions — 533
M. Comini, G. Levi, and M.C. Meo

Curbing Theories: Fixpoint Semantics and Complexity Issues — 545
F. Scarcello, N. Leone, and L. Palopoli

CONSTRAINTS

A Dataflow Semantics for Constraint Logic Programs — 557
L. Colussi, E. Marchiori, and M. Marchiori

Labeling in CLP(FD) with Evolutionary Programming — 569
A. Ruiz-Andino Illera, and J.J. Ruz Ortiz

Constraint Systems for Pattern Analysis of Constraint Logic-Based Languages — 581
R. Bagnara

Tuple Inheritance: A New Kind of Inheritance for (Constraint) Logic Programming — 593
J.J. Moreno-Navarro, J. García-Martín, and A. del Pozo-Pietro

ANALYSIS

Declarative Diagnosis Revisited — 607
M. Comini, G. Levi, and G. Vitiello

"Optimal" Collecting Semantics for Analysis in a Hierarchy of Logic Program Semantics — 619
R. Giacobazzi

Contributions to a Theory of Existential Termination for Definite Logic Programs — 631
G. Levi, and F. Scozzari

A Case Study in Logic Program Verification: the Vanilla Metainterpreter — 643
D. Pedreschi, and S. Ruggieri

INVITED LECTURES

important for making log programming
nearer to specifications.

Arrays, Bounded Quantification and Iteration in Logic and Constraint Logic Programming

Krzysztof R. Apt
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
Department of Mathematics and Computer Science
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

Abstract

We claim that programming within the logic programming paradigm suffers from lack of attention given to iteration and arrays. To convince the reader about their merits we present several examples of logic and constraint logic programs which use iteration and arrays instead of explicit recursion and lists. These programs are substantially simpler than their counterparts written in the conventional way. They are easier to write and to understand, are guaranteed to terminate and their declarative character makes it simpler to argue about their correctness. Iteration is implemented by means of bounded quantification.

1 Introduction

Any systematic course on programming in the imperative style (say using Pascal), first concentrates on iteration constructs (say **while** or **repeat**) and only later deals with recursion. Further, the data structures are explained first by dealing with the static data structures (like arrays and records) and only later with the dynamic data structures (which are constructed by means of pointers).

In the logic programming framework the distinctions between iteration and recursion, and between static and dynamic data structures are lost. One shows that recursion is powerful enough to simulate iteration and rediscovers the latter by identifying it with tail recursion. Arrays do not exist. In contrast, records can be modelled by terms, and dynamic data structures can be defined by means of clauses, in a recursive fashion (with the exception of lists for which in Prolog there is support in the form of built-ins and a more friendly notation).

One of the side effects of this approach to programming is that one often uses a sledgehammer to cut the top of an egg. Even worse, simple problems have unnecessarily complex and clumsy solutions in which recursion is used when a much easier solution using iteration exists, is simpler to write and understand, and — perhaps even more important — is closer to the original specification.

In this paper we would like to propose an alternative approach to programming in logic programming and in constraint logic programming — an approach in which adequate stress is put on the use of arrays and iteration. Because iteration can be expressed by means of bounded quantification, a purely logical construct, the logic programming paradigm is not "violated". On the contrary, it is enriched, clarified and better tailored for the programming needs.

Arrays are especially natural when dealing with vectors and matrices. The use of dynamic data structures to write programs dealing with such objects is unnatural. We shall try to illustrate this point by presenting particularly simple solutions to problems such as the n-queens problem, the knight's tour, the map colouring problem, the cutting stock problem, and other problems involving backtracking.

Further, by adding to the language operators which allow us to express optimization, i.e. minimization and maximization, we can easily write programs for various optimization problems.

For pedagogical reasons we limit our attention to programs that involve iteration and optimization constructs. Of course, explicit recursion has its place both in logic programming and in constraint logic programming. One of the main purposes of this paper is to illustrate how much can be achieved without it.

In the programs considered in this paper recursion is hidden in the implementation of the bounded quanfiers and this use of recursion is guaranteed to terminate. Consequently, these programs always terminate. As termination is one of the major concerns in the case of logic programming, from the correctness point of view it is better to use iteration instead of recursion, when a choice arises. Also, iteration can be implemented more efficiently than recursion (see Barklund and Bevemyr [BB93] for an explanation how to extend WAM to implement iteration in Prolog).

This work has a preliminary character and can be seen as an attempt to identify the right linguistic concepts which simplify programming in the logic programming paradigm. When presenting this view of programming within the logic programming paradigm we were very much influenced by the publications of Barklund and Millroth [BM94], Voronkov [Vor92] and Kluźniak [Klu93]. In fact, the constructs whose use we advocate, i.e. bounded quantification and arrays, were already proposed in these papers. The only, possibly new, contribution of this paper is a suggestion to include these constructs in constraint logic programming.

2 Bounded Quantifiers

Bounded quantifiers in logic programming were introduced in Kluźniak [Klu91] and are thoroughly discussed in Voronkov [Vor92] (where also earlier references in Russian are given). They are also used in Kluźniak [Klu93] (see also Kluźniak and Miłkowska [KM94]) in a specification language SPILL-2 in which executable specifications can be written in the logic programming style.

Following Voronkov [Vor92] we write them as ∃X ∈ L Q (the bounded existential quantifier) and ∀X ∈ L Q (the bounded universal quantifier), where L is a list and Q a query, and define them as follows:

```
∃X ∈ [Y | Ys] Q  ←  Q{X/Y}.
∃X ∈ [Y | Ys] Q  ←  ∃X ∈ Ys Q.
∀X ∈ [Y | Ys] Q  ←  Q{X/Y}, ∀X ∈ Ys Q.
∀X ∈ [] Q.
```

∀B(X, [Y|Ys], Q) ← sub(Q, X, Y),
∀B(X, Ys, Q).
∀B(X, [], Q).

Voronkov [Vor92] also discusses two other bounded quantifiers, written as ∃X ⊏ L Q and ∀X ⊏ L Q, where X ⊏ L is to be read "X is a suffix of L", which we do not consider here.

To some extent the use of bounded quantifiers allows us to introduce in some compact form the "and" and the "or" branching within the program computations. This reveals some connections with the approach of Harel [Har80], though we believe that the expressiveness and ease of programming within the logic programming paradigm makes Harel's programming proposal obsolete.

Even without the use of arrays the gain in expressiveness achieved by means of bounded quantifiers is quite spectacular. Consider for example the following problem.

Problem 1 Write a program which tests whether one list is a subset of another.

Solution

```
subset(Xs, Ys)  ←  ∀X ∈ Xs ∃Y ∈ Ys X = Y.
```

Several other examples can be found in Voronkov [Vor92]. Here we content ourselves with just one more, in which we use delay declarations very much like in modern versions of Prolog, (for example in ECL^{i}PS^e) or the programming language Gödel of Hill and Lloyd [HL94]).

Problem 2 Write a program checking the satisfiability of a Boolean formula.

Solution We assume here that the input Boolean formula is written using Prolog notation, so for example (¬ X, Y) ; Z stands for (¬ X ∧ Y) ∨ Z.

```
sat(X)  ←  X, generate(X).
generate(X)  ←  vars(X, Ls), ∀Y ∈ Ls ∃Z ∈ [true, fail] Y = Z.
DELAY X UNTIL nonvar(X).
```

Comments This remarkably short program uses meta-variables and a mild extension of the delay declarations to meta-variables. The delay declaration used here delays any call to a meta-variable until it becomes instantiated. `vars(t, Ls)` for a term `t` computes in `Ls` the list of the variables occurring in `t`. Its definition is omitted. `vars(X, Ls)` can be easily implemented using the `var(X)` and `univ` built-in's of Prolog. `true` and `fail` are Prolog's built-in's.

In this program it is not advisable to delay the calls to negative literals until they become ground. Such a delay would reduce checking for satisfiability of subformulas which begin with the negation sign to a naive generate and test method.

Even though this program shows the power of Prolog, we prefer to take another course and use types instead of exploring extensions of Prolog, which is an untyped language.

3 Arrays and Bounded Quantifiers in Logic Programming

Arrays in logic programming were introduced in Eriksson and Rayner [ER84]. Barklund and Bevemyr [BB93] proposed to extend Prolog with arrays and studied their use in conjunction with the bounded quantification. In our opinion the resulting extension (unavoidably) suffers from the fact that Prolog is an untyped language. In Kluźniak [Klu93] arrays are present, as well, where they are called indexable sequences.

More recently, Barklund and Hill [BH95] proposed to add arrays and restricted quantification, a generalization of the bounded quantification, to Gödel, the programming language which does use types.

In the programs below we use bounded quantification, arrays and type declarations. The use of bounded quantifiers and arrays makes them simpler, more readable and closer to specifications. We declare constants, types, variables and relations in a style borrowed from the programming language Pascal. The choice of notation is preliminary.

We begin with two introductory examples.

Problem 3 Check whether a given sequence of 100 integers is ordered.

Solution

```
const n = 100.
rel ordered: array [1..n] of integer.
ordered(A) ← ∀I ∈ [1..n-1] A[I] ≤ A[I+1].
```

It seems that the ∃quantifier solves the problem of the bad nesting of if – then construct together with backtracking

Comments This example shows that within the array subscripts terms should be evaluated, so that we can identify `1+1` with `2` etc. More precisely, "+" should be viewed here as an external procedure in the sense of Małuszyński et al. [MBB+93].

Note that the bounded universal quantifier `∀I ∈ [1..n]` does *not* correspond to the imperative **for** `i:=1` **to** `n` loop. The former is executed as long as a failure does not arise, i.e. up to n times, whereas the latter is executed precisely n times. The programming construct `∀I ∈ [1..n] Q` actually corresponds to the construct

```
for i:=1 to n do if ¬ Q then
                begin
                  failure := true; exit
                end
```

which is clumsy and unnatural within the imperative programming paradigm.

(Feliks Kluźniak suggested to us the following, slightly more natural interpretation of `∀I ∈ [1..n] Q`:

```
i:=1;
while i ≤ n cand Q do i:=i+1;
failure := i ≤ n,
```

where **cand** is the "conditional **and**" connective (see Gries [Gri81, pages 68-70].))

Problem 4 Generate all members of a given sequence of 100 elements.

Solution

```
const n = 100.
rel member: (*, array [1..n] of *).
member(X, Y) ← ∃I ∈ [1..n] X = Y[I].
```

Comments Here, `Y` is the given sequence. "*" stands for an unknown type. "=" is a built-in declared as

```
rel =: (*, *).
DELAY X = Y UNTIL known(X) ∨ known(Y).
```

In other words, "=" is defined on any type and the calls to "=" are delayed until the value of one of its arguments is known, i.e. uniquely determined. If the values of both arguments are known, then it behaves like the usual comparison relation of Prolog and if the value of only one argument is known and the other is a, possibly subscripted, variable, then "=" behaves like the `is` built-in of Prolog. The case when one of the arguments is known and the other is not a variable does not arise here. `known(X)` is a built-in which holds when its argument is uniquely determined. It corresponds to `ground(X)` in Prolog.

This example shows the usefulness of polymorphic types in the presence of arrays. The bounded existential quantifier `∃I ∈ [1..n]` implements backtracking and has no counterpart within the imperative programming paradigm.

Problem 5 Arrange three 1's, three 2's, ..., three 9's in sequence so that for all $i \in [1, 9]$ there are exactly i numbers between successive occurrences of i (see Coelho and Cotta [CC88, page 193]).

Solution

```
rel sequence:  array [1..27] of [1..9].
sequence(A)  ←  ∀I ∈ [1..9] ∃J ∈ [1..25-2I]
    (A[J] = I, A[J+I+1] = I, A[J+2I+2] = I)).
```

Comments The range `J ∈ [1..25-2I]` comes from the requirement that the indices `J, J+I+1, J+2I+2` should lie within `[1..27]`. Thus `J+2I+2` $\leq$ `27`, that is `J` $\leq$ `25-2I`.

Problem 6 Generate all permutations of a given sequence of 100 elements.

First we provide a solution for the case when there are no repeated elements in the sequence.

Solution 1

```
const n = 100.
rel permutation:  (array [1..n] of *, array [1..n] of *).
permutation(X, Y)  ←  ∀I ∈ [1..n] ∃J ∈ [1..n] Y[J] = X[I].
```

Here, `X` is the given sequence. Alternatively,

```
permutation(X, Y)  ←  ∀I ∈ [1..n] member(X[I], Y).
```

Comments Note the similarity in the structure between this program and the one that solves problem 1. This program is incorrect when the sequence contains repeated elements. For example for `n = 3` and `X:= 0,0,1, Y:= 0,1,1` is a possible answer.

To deal with the general case we use local array declarations and reuse the above program.

Solution 2

```
const n = 100.
rel permutation:  (array [1..n] of *, array [1..n] of *).
permutation(X, Y)  ←
    var A: array [1..n] of [1..n].
    ∀I ∈ [1..n] ∃J ∈ [1..n] A[J] = I,
    ∀I ∈ [1..n] Y[I] = X[A[I]].
```

in Relog, X ≠ 0 [illegible] error if X is a variable
in CCP is to take 0 away from the domain of X

Comments This solution states that `A` is an onto function from `[1..n]` to `[1..n]` and that a permution of a sequence of `n` elements is obtained by applying the function `A` to its indices.

Next, consider two well-known chess puzzles.

Problem 7 Place 8 queens on the chess board so that they do not check each other.

First, we provide a naive generate and test solution. It will be of use in the next section.

Solution 1

```
const n = 8.
type board:  array [1..n] of [1..n].
rel queens, generate, safe:  board.

queens(X)  ←  generate(X), safe(X).

generate(X)  ←  ∀I ∈ [1..n] ∃J ∈ [1..n] X[I] = J.

safe(X)  ←  ∀I ∈ [1..n] ∀J ∈ [I+1..n]
    (X[I] ≠ X[J], X[I] ≠ X[J] + (J-I), X[I] ≠ X[J] + (I-J)).
```

Comments To improve readability `board` is explicitly declared here as a type. Declaratively, this program states the conditions which should be satisfied by the values chosen for the queens. "≠" is a built-in declared as

```
rel ≠:  (*, *).
```

In this section we use it only to compare terms with known values. Then "≠" behaves like the usual arithmetic inequality relation of Prolog. A more general usage of "≠" will be explained in the next section.

Next, we give a solution which involves backtracking.

Solution 2

```
const n = 8.
type board:  array [1..n] of [1..n].
rel queens:  board.

queens(X)  ←  ∀J ∈ [1..n] ∃K ∈ [1..n]
    (X[J] = K,
     ∀I ∈ [1..J-1]
        (X[I] ≠ X[J], X[I] ≠ X[J] + (J-I), X[I] ≠ X[J] + (I-J))).
```

Comments Declaratively, this program states the conditions each possible value K for a queen placed in column J should satisfy.

Problem 8 Knight's tour. Find a cyclic route of a knight on the chess board so that each field is visited exactly once.

Solution We assign to each field a value between 1 and 64 and formalize the statement "from every field there is a "knight-reachable" field with the value one bigger". By symmetry we can assume that the value assigned to the field X[1, 1] is 1. Taking into account that the route is to be cyclic we actually get the following solution.

```
const n = 8.
type board:  array [1..n, 1..n] of [1..n²].
rel knight:  board.
    go_on:  (board, [1..n], [1..n]).

knight(X)  ←  ∀I ∈ [1..n] ∀J ∈ [1..n] go_on(X, I, J), X[1, 1] = 1.

go_on(X, I, J)  ←  ∃I1 ∈ [1..n] ∃J1 ∈ [1..n]
     (abs((I-I1)·(J-J1)) = 2, X[I1, J1] = (X[I, J] mod n²) + 1).

DELAY go_on(X, I, J) UNTIL known(X[I,J]).
```

Comments Note that the equation abs(X · Y) = 2 used in the definition of go_on has exactly 8 solutions, which determine the possible directions for a knight move. Observe that each time this call to "=" is selected, both arguments of it are known. The efficiency of go_on could of course be improved by explicitly enumerating the choices for the offsets of the new coordinates w.r.t. the old ones.

The behaviour of the above program is quite subtle. First, thanks to the delay declaration, 64 constraints of the form go_on(X, I, J) are generated. Then, thanks to the statement X[1, 1] = 1, the first of them is "triggered" which one by one activates the remaining constraints. The backtracking is carried out by choosing different values for the variables I1 and J1. The delay declaration is not needed, but without it this program would be hopelessly inefficient.

It is interesting to note that in Wirth [Wir76], a classical book on programming in Pascal, the solutions to the last two problems are given as prototypical examples of recursive programs. Here recursion is implicit.

We conclude this section by one more program. It will be needed in the next section.

Problem 9 Let m = 50 and n = 100. Determine the number of different elements in an array X:**array** [1..m, 1..n] **of integer**.

Solution

```
const m = 50.
      n = 100.
type board:  array [1..m,1..n] of integer.
rel count:  (board, natural).

count(X, Number) ←
    Number = m · n -
     #(I, J: I ∈ [1..m], J ∈ [1..n]:
         (∃K ∈ [1..I-1] ∃L ∈ [1..n] X[I,J] = X[K,L])
           % X[I,J] occurs in an earlier row
          ∨ (∃L ∈ [1..J-1] X[I,J] = X[I,L]).
           % X[I,J] occurs earlier in the same row
      ).
```

Comments In this program we used the counting quantifier introduced in Gries [Gri81, page 74] and adopted in Kluźniak [Klu93] in the specification language SPILL-2. In general, given lists L1, L2, the term #(I, J: I ∈ L1, J ∈ L2: Q) stands for the number of pairs (i,j) such that i ∈ L1, j ∈ L2 and for which the query Q{I/i,J/j} succeeds. It is possible to avoid the use of the counting quantifier at the expense of introducing a local array of type board. This alternative program is more laborious to write.

This concludes our presentation of selected logic programs written using arrays and bounded quantifiers. Other examples, including those involving numerical computation can be found in Barklund and Millroth [BM94].

4 Arrays and Bounded Quantifiers in Constraint Logic Programming

We now present some constraint logic programs. These are constraint programs with finite domains in the style of van Hentenryck [vH89]). Each of them has a similar pattern: constraints are first generated, and then resolved after the possible values for variables are successively generated. To clarify their use we provide here alternative solutions to two problems discussed in the previous section.

Problem 10 Solve problem 7 by means of constraints.

Solution

```
const n = 8.
type board:  array [1..n] of [1..n].
rel queens, safe, generate:  board.
```

in logic programming: generate & test
in constraint log. programming: first generate constraints & then the value
[illegible]

```
queens(X) ← safe(X), generate(X).
safe(X) ← ∀I ∈ [1..n] ∀J ∈ [I+1..n]
    (X[I] ≠ X[J], X[I] ≠ X[J] + (J-I), X[I] ≠ X[J] + (I-J)).
generate(X) ← ∀I ∈ [1..n] ∃J ∈ dom(X[I]) X[I] = J.
```

Comments Here dom(X), for a (possibly subscripted) variable X, is a built-in which denotes the list of current values in the domain of X, say in the ascending order. The value of dom(X) can change only by decreasing, by executing a constraint, so in the above program an atom of the form X ≠ t.

The relation "≠" was used in the previous section only in the case when both arguments of it were known. Here we generalizes its usage, as we now allow that one or both sides of it are not known. In fact, "≠" is a built-in defined as in van Hentenryck [vH89, pages 83-84], though generalized to arbitrary non-compound types.

We require that one of the following holds:

- Both sides of "≠" are known. This case is explained in the previous section.
- At most one of the sides of "≠" is known and one of the sides of "≠", denoted below by X, is either a simple variable or a subscripted variable with a known subscript.

In the second case X ≠ t is defined as follows, where for a term s, Val(s) stands for the set of its currently possible values:

```
if Val(X) ∩ Val(t) = ∅ then succeed
elseif Val(t) is a singleton then % t is known, so X is not known
    begin dom(X) := dom(X) - Val(t); % dom(X) ≠ ∅
      if dom(X) = {f} then X := f
    end .
```

If neither Val(X) ∩ Val(t) = ∅ nor Val(t) is a singleton, then the execution of X ≠ t is delayed. We treat t ≠ X as X ≠ t.

So for example in the program fragment

```
...
type bool:  [false, true].
var B: bool.
    A: array [1..2] of bool.
A[1] ≠ A[2], A[1] ≠ B, B = true.
...
```

the constraints A[1] ≠ A[2] and A[1] ≠ B are first delayed and upon the execution of the atom B = true the variable A[1] becomes false and A[2] becomes true.

[illegible]

In turn, in the case of the program given above the execution of an atom of the form X[I] = J for some I,J ∈ [1..n] can affect the domains of the variables X[K] for K ∈ [I+1..n]

This solution to the 8 queens problem is a forward checking program (see van Hentenryck [vH89, pages 122-127]). Note the textual similarity between this program and the one given in solution 1 to problem 7. Essentially, the calls to the safe and generate relations are now reversed. The generate relation corresponds to the labeling procedure in van Hentenryck [vH89]). In the subsequent programs the definition of the generate relation is always of the same format and is omitted.

Problem 11 Solve problem 6 by means of constraints.

Solution

```
const n = 100.
rel permutation:  (array [1..n] of *, array [1..n] of *).
permutation(X, Y) ←
    type board:  array [1..n] of [1..n].
    rel one_one, generate:  board.
    one_one(Z) ← ∀I ∈ [1..n] ∀J ∈ [I+1..n] Z[I] ≠ Z[J].
    var A: board.
    one_one(A), generate(A),
    ∀I ∈ [1..n] Y[I] = X[A[I]].
```

Comments In this solution, apart from the local array declaration, we also use local type and relation declarations. The efficiency w.r.t. to the logic programming solution is increased by stating, by means of the call to the one_one relation, that A is a 1-1 function. This replaces the previously used statement that A is an onto function. The call to one_one generates $n \cdot (n-1)/2 = 4950$ constraints.

We conclude this section by dealing with another classic problem — that of colouring a map.

Problem 12 Given is a binary relation neighbour between countries. Colour a map in such a way that no two neighbours have the same color.

Solution

```
type color:  [blue, green, red, yellow].
    countries:  [austria, belgium, france, italy, ...].
rel map_color, constrain, generate:  array countries of color.
    neighbour:  (country, country).
map_color(X) ← constrain(X), generate(X).
```

```
constrain(X) ← ∀I ∈ countries ∀J ∈ countries
    neighbour(I,J) → X[I] ≠ X[J].
```

Comments We interpret here `P → Q` as follows:

```
(P → Q)  ←  P, Q.
(P → Q)  ←  ¬P.
```

so like the `IF P THEN Q` statement of Gödel. Note that in the above program at the moment of selection of the `P → Q` statement `P` is ground. Obviously, an efficient implementation of `P → Q` should avoid the reevaluation of `P`.

Thus the `constrain` relation generates here the constraints of the form `X[I] ≠ X[J]` for all `I,J` such that `neighbour(I,J)`.

5 Adding Minimization and Maximization

Next, we introduce a construct allowing us to express in a compact way the requirement that we are looking for an optimal solution. To this end we introduce the *minimization operator* `Y = μX:Q` which is defined as follows:

```
Y = μX:Q  ←  Q{X/Y}, ¬(∃X ( X < Y, Q)).
```

We assume here that `X` and `Y` are of the same type and that $<$ is a built-in ordering on the domain of the type of `X` and `Y`. The existential quantifier `∃X Q` is defined by the clause

```
∃X Q  ←  Q.
```

The efficient implementation of the minimization operator should make use of memoization (sometimes called tabulation) to store the solutions to the query Q found during the successive attempts to find a minimal one.

A dual operator, the *maximization operator* `Y = νX:Q`, is defined by:

```
Y = νX:Q  ←  Q{X/Y}, ¬ (∃X ( X > Y, Q)).
```

As before we assume that $>$ is a built-in ordering on the domain of the type of `X` and `Y`. In Barklund and Hill [BH95] the minimization and the maximization operators are introduced as a form of arithmetic quantifiers, in the style of the counting quantifier introduced earlier. The above two clauses show that they are derived concepts.

The following simple example illustrates the use of these constructs.

Problem 13 Find a minimum and a maximum of a given sequence of 100 integers.

Solution

```
const n = 100.
rel min_and_max:  (integer, integer, array [1..n] of integer).
min_and_max(Min, Max, A)  ←
    Min = μX: ∃I ∈ [1..n] X = A[I],
    Max = νX: ∃I ∈ [1..n] X = A[I].
```

Next, we use these two operators in two constraint programs.

Problem 14 The cutting stock problem (see van Hentenryck [vH89, pages 181-187]). There are 72 configurations, 6 kinds of shelves and 4 identical boards to be cut. Given are 3 arrays:

```
Shelves:array [1..72, 1..6] of natural,
Req:array [1..6] of natural,
Waste:array [1..72] of natural.
```

`Shelves[K,J]` denotes the number of shelves of kind `J` cut in configuration `K`, `Waste[I]` denotes the waste per board in configuration `I` and `Req[J]` the required number of shelves of kind `J`. The problem is to cut the required number of shelves of each kind in such a way that the total waste is minimized.

Solution We represent the chosen configurations by the array
`Conf: array [1..4] of [1..72]`
where `Conf[I]` denotes the configuration used to cut the board `I`.

```
rel solve:  (array [1..4] of [1..72], natural).
  generate:  array [1..4] of [1..72].

solve(Conf, Sol)  ←
    Sol = μTCost:
        % Sol is the minimal TCost such that:
    ∀I ∈ [1..3] Conf[I] ≤ Conf[I+1],
        % symmetry between the boards
    ∀J ∈ [1..6] $\Sigma_{I=1}^{4}$ Shelves[Conf[I],J] ≥ Req[J],
        % enough shelves are cut
    TCost = $\Sigma_{I=1}^{4}$ Waste[Conf[I]],
        % TCost is the total waste
    generate(Conf).
```

Comments In this program we used as a shorthand the sum notation "Σ ...". In general, it is advisable to use the sum quantifier (see Gries [Gri81, page 72]), which allows us to use $\Sigma_{I=k}^{l}$ `t` as a term. The sum quantifier is adopted in SPILL-2 language of Kluźniak [Klu93]. Kluźniak's notation for this expression is: `(S I: k`

$\leq$ I $\leq$ l: t). The interpretation of the constraints of the form X $\leq$ t, X $\geq$ t or X = t is similar to that of X $\neq$ t and is omitted.

We conclude by solving the following problem.

Problem 15 Let m = 50 and n = 100. Given is an array Co which assigns to each pixel on an m by n board a colour. A *region* is a maximal set of adjacent pixels that have the same colour. Determine the number of regions.

In the program below we assign to the pixels belonging to the same region the same natural number, drawn between 1 and m·n. If we maximize the number of so used natural numbers we obtain the desired solution.

Solution

```
const m = 50.
      n = 100.
type color:  [blue, green, red, yellow].
     pattern:  array [1..m,1..n] of color.
     board:  array [1..m,1..n] of [1..m·n].
rel pixel:  (pattern, natural).
    no:  (pattern, board).
    generate:  board.
    count:  (board, natural).
pixel(Co, Sol)  ←  Sol = νNumber:
    var X: board.
    no(Co, X), generate(X), count(X, Number).
no(Co, X)  ←  ∀I ∈ [1..m] ∀J ∈ [1..n]
    (
     (I < m  →  (Co[I,J] = Co[I+1,J]  ↔  X[I,J] = X[I+1,J])),
     (J < n  →  (Co[I,J] = Co[I,J+1]  ↔  X[I,J] = X[I,J+1]))
    ).
```

Comments The count relation is defined in the solution to problem 9. In the above program first $2m \cdot n - (m + n) = 9850$ constraints are generated. Each of them deals with two adjacent fields and has the form of an equality or inequality. Then the possible values for the elements of X are generated and the number Number of so used natural numbers is counted. The maximum value for Number is then the desired solution.

The resulting program is probably not efficient, but still it is interesting to note that the problem at hand can be solved in a simple way without explicit recursion.

6 Conclusions

We have presented here several logic and constraint logic programs that use bounded quantification and arrays. We hope that these examples convinced the readers about the usefulness of these constructs. We think that this approach to programming is especially attractive when dealing with various optimization problems, as their specifications often involve arrays, bounded quantification, summation, and minimization and maximization. Constraint programming solutions to these problems can be easily written using arrays, bounded quantifiers, the sum and cardinality quantifiers, and the minimization and maximization operators. As examples let us mention the stable marriage problem, various timetabling problems and integer programming.

Of course, it is not obvious whether the solutions so obtained are efficient. We expect, however, that after an addition of a small number of built-in's, like **deleteff** and **deleteffc** of van Hentenryck [vH89, pages 89-90], it will be possible to write simple constraint programs which will be comparable in efficiency with those written in other languages for constraint logic programming.

When introducing arrays we were quite conservative and only allowed static arrays, i.e. arrays whose bounds are determined at compile time. Of course, in a more realistic language proposal also open arrays, i.e. arrays whose bounds are determined at run-time should be allowed. One might also envisage the use of flexible arrays, i.e. arrays whose bounds can change at run-time.

In order to make this programming proposal more realistic one should provide a smooth integration of arrays with recursive types, like lists and trees. In the language SPILL-2 of Kluźniak [Klu93] types are present but only as sets of ground terms, and polymorphism is not allowed. Barklund and Hill [BH95] proposed to add arrays to Gödel (which does support polymorphism) as a system module. We would prefer to treat arrays on equal footing with other types.

We noticed already that within the logic programming paradigm the demarkation line between iteration and recursion differs from the one in the imperative programming paradigm. In order to better understand the proposed programming style one should first clarify when to use iteration instead of recursion. In this respect it is useful to quote the opening sentence of Barklund and Millroth [BM94]: "Programs operating on inductively defined data structures, such as lists, are naturally defined by recursive programs, while programs operating on "indexable" data structures, such as arrays, are naturally defined by iterative programs".

We do not entirely agree with this remark. For example, the "suffix" quantifiers mentioned in Section 2 allow us to write many list processing programs without explicit use of recursion (see Voronkov [Vor92]) and the **quicksort** program written in the logic programming style is more natural when written using recursion than iteration.

The single assignment property of logic programming makes certain programs that involve arrays (like Warshall's algorithm) obviously less space efficient than their imperative programming counterparts. This naturally motivates research on efficient implementation techniques of arrays within the logic programming paradigm.

Finally, a comment about the presentation. We were quite informal when explaining the meaning of the proposed language constructs. Note that the usual definition of SLD-resolution has to be appropriately modified in presence of arrays and bounded quantification. For example, the query `X[1] = 0, ∀I ∈ [1..2] X[I]` $\neq$ `0` fails but this fact can be deduced only when the formation of resolvents is formally explained. To this end substitution for subscripted variables needs to be properly defined. One possibility is to adopt one of the definitions used in the context of verification of imperative programs (see Apt [Apt81, pages 460-462]). We leave the task of defining a formal semantics to another paper.

Acknowledgements I would like to thank here Jonas Barklund and Feliks Kluźniak for useful discussions on the subject of bounded quantification and Pascal van Hentenryck for encouragement at the initial stage of this work. Also, I am grateful to Feliks Kluźniak for helpful comments on this paper.

References

[Apt81] K.R. Apt. Ten years of Hoare's logic, a survey, part I. *ACM TOPLAS*, 3:431–483, 1981.

[BB93] J. Barklund and J. Bevemyr. Prolog with arrays and bounded quantifications. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning—Proc. 4th Intl. Conf.*, LNCS 698, pages 28–39, Berlin, 1993. Springer-Verlag.

[BH95] J. Barklund and P. Hill. Extending Gödel for expressing restricted quantifications and arrays. UPMAIL Tech. Rep. 102, Computer Science Department, Uppsala University, Uppsala, 1995.

[BM94] J. Barklund and H. Millroth. Providing iteration and concurrency in logic programs through bounded quantifications. UPMAIL Tech. Rep. 71, Computer Science Department, Uppsala University, Uppsala, 1994.

[CC88] H. Coelho and J. C. Cotta. *Prolog by Example*. Springer-Verlag, Berlin, 1988.

[ER84] L.-H. Eriksson and M. Rayner. Incorporating mutable arrays into logic programming. In S. Å. Tarnlund, editor, *Proceedings of the 1991 International Conference on Logic Programming*, pages 101–114. Uppsala University, 1984.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[Har80] D. Harel. And/or programs: a new approach to structured programming. *ACM Toplas*, 2(1):1–17, 1980.

[HL94] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994.

[Klu91] F. Kluźniak. Towards practical executable specifications in logic. Research report LiTH-IDA-R-91-26, Department of Computer Science, Linköping University, August 1991.

[Klu93] F. Kluźniak. SPILL-2: the language. Technical report ZMI Reports No 93-03, Institute of Informatics, Warsaw University, July 1993. A deliverable for year 1 of the BRA Esprit Project Compulog 2.

[KM94] F. Kluźniak and M. Miłkowska. Readable, runnable requirements specifications: Bridging the credibility gap. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings of the 6th International Symposium, PLILP'94. Madrid, September 1994*, pages 449–450. Springer-Verlag, 1994.

[MBB$^+$93] J. Maluszyński, S. Bonnier, J. Boye, F. Kluźniak, A. Kågedal, and U. Nilsson. Logic programs with external procedures. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Current Trends in Logic Programming Languages*, pages 21–48. The MIT Press, Cambridge, Massachussets, 1993.

o [vH89] 122-127 P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[Vor92] A. Voronkov. Logic programming with bounded quantifiers. In A. Voronkov, editor, *Logic Programming and Automated Reasoning—Proc. 2nd Russian Conference on Logic Programming*, LNCS 592, pages 486–514, Berlin, 1992. Springer-Verlag.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

Completeness in Abstract Interpretation [*]

(Invited talk)

Patrick COUSOT

LIENS - DMI
École Normale Supérieure
45 rue d'Ulm
75230 Paris cedex 05 (France)
cousot@dmi.ens.fr

Abstract interpretation [1] is a method for designing hierarchies of semantics as well as specifications of program analyzers by approximation of these semantics. Because of undecidability problems such as the termination problem, abstract interpretation based program analysis methods are fundamentally incomplete. Moreover implementation techniques such as the use of widenings/narrowing to speed up convergence of iterative fixpoint computation methods give the impression that the result of the analysis performed by the abstract interpreter is not at all predictable by the user.

This is in contrast with methods such as set-based analysis à la Heinze or type inference à la Milner which look different from abstract interpretation, for which numerous completeness results have been published and for which the result of the analysis can be predicted by the user, at least in principle, through the use of a rule-based inference system.

It has been shown recently that both set-based analysis [2] and type-inference [3] are abstract interpretations. Set-based analysis uses a finite abstract symbolic domain for each particular program (although it is an infinite domain when considering all possible programs). The unification based type-inference algorithm uses an infinite abstract domain together with a rather naïve widening operator (which may not look natural to some users).

[*]This work was partly supported by ESPRIT BRA 8130 LOMAPS.

This clearly shows that when one speaks of the fundamental incompleteness of abstract interpretation in contrast with the relative completeness of type inference systems, one cannot speak of the exactly same notions.

After a brief introduction to basic abstract interpretation notions, the purpose of the talk is to solve this apparent contradiction by eliminating superfluous differences in presentation of program analysis methods and by introducing a hierarchy of different and partially comparable notions of completeness. This explains the various acceptations of the notion with regard to fixpoint inference/fusion, computer representation of the abstract domain, computability of the abstract property transformer, (iterative) fixpoint computation, rule-based inference algorithm, convergence acceleration, etc. Numerous examples are provided in the context of logic programming with a few incursions in functional programming.

References

[1] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Prog.*, 13(2–3):103–179, 1992. (The editor of JLP has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.ens.fr/~cousot`.)

[2] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc. 7th FPCA*, pages 170–181, La Jolla, Calif., 25–28 June 1995. ACM Press.

[3] B. Monsuez. Polymorphic types and widening operators. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Proc. 3rd International Workshop WSA'93 on Static Analysis*, Padova, (I), LNCS 724, pages 267–281. Springer-Verlag, 22–24 Sept. 1993.

Logical foundations for multi-agent systems

Robert Kowalski

Imperial College - Department of Computing

180 Queen's Gate, London SW7 2BZ, UK

I will summarise the current status of my work with Francesca Toni, Fariba Sadri, Jacinto Davila, Ber Permpoontanalarp, Eric Fung and Gerhard Wetzel on developing logical foundations for multi-agent systems.

The core of these foundations is a new approach to logic programming which unifies abductive logic programming and constraint logic programming. This approach allows predicates to be defined in the usual logic programming manner, augmented with integrity constraints, which are properties of the definitions. Predicates are executed backwards using the definitions, as well as forwards using the integrity constraints. The approach is being developed both to serve as the inference engine for individual agents and as a programming language paradigm in its own right. Applications of the approach to operations research problems are also being investigated.

Integrity constraints are also used to obtain activity and reactivity in individual agents. Observations, which update the knowledge base of an agent, are checked for consistency with the integrity constraints. Integrity checking generates new goals, some of which may be converted into actions to be executed by the agent.

The overall observation-reasoning-action cycle is controlled by a resource-bounded metalogic program. The resource bound allows the reasoning and planning component of the cycle to be interrupted at any time to obtain an executable approximation to a plan which achieves the agent's goals. The representation of actions and temporal relationships is formulated in a version of the event calculus.

An agent's plans can contain actions to be performed by the agent itself, as well as actions to be performed by other agents. Moreover, actions can be speech acts, in general, and can convey information or requests from one agent to another, in particular. Agents can use such speech acts to coordinate their actions. We have begun to investigate the use of argumentation theory to provide a framework for such speech acts . In addition, we intend to investigate the applicability of concepts from deontic logic (the logic of obligation, prohibition and permission) to the problem of regulating interaction among agents.

On the Abstract Diagnosis of Logic Programs

Marco Comini, Giorgio Levi
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
{comini,levi}@di.unipi.it

Giuliana Vitiello
Dipartimento di Informatica ed Applicazioni
Università di Salerno, Baronissi (Salerno), Italy
giuvit@udsab.dia.unisa.it

Abstract

Abstract diagnosis of logic programs is an extension of declarative diagnosis, where we deal with specifications of operational properties, which can be characterized as abstractions of *SLD*-trees (observables).
We introduce a simple and efficient method to detect incompleteness errors, which is based on the application of the immediate consequences operator to the specification. The method is proved to be correct and complete whenever the immediate consequences operator has a unique fixpoint. We prove that this property is always satisfied if the program belongs to a large class of programs (acceptable programs). We then show that the same property can be proved for any program P, if the observable belongs to a suitable class of observables. We finally consider the problem of diagnosis of incompleteness for a weaker class of observables, which are typical of program analysis.

1 Introduction

Abstract diagnosis [9, 11] is a combination of three known techniques, i.e., *algorithmic (declarative) diagnosis (debugging)* [25, 18, 21, 15], the *s-semantics approach* to the definition of program denotations modeling various observable behaviors [16, 17, 20, 4, 3], and *abstract interpretation* [12, 13, 14].

The diagnosis problem can formally be defined as follows. Let P be a program, $[\![P]\!]_\alpha$ be the behavior of P w.r.t. the observable property α, and $\mathcal{I}_\alpha$ be the specification of the *intended* behavior of P w.r.t. α. The diagnosis consists of comparing

$[\![P]\!]_\alpha$ and $\mathcal{I}_\alpha$ and determining the "errors" and the program components which are sources of errors, when $[\![P]\!]_\alpha \neq \mathcal{I}_\alpha$.

The above formulation is parametric w.r.t. the *observable* α, which is considered in the specification $\mathcal{I}_\alpha$ and in the actual behavior $[\![P]\!]_\alpha$.

Declarative diagnosis is concerned with model-theoretic properties rather than with the operational behavior. The specification is therefore the intended declarative semantics of the program, which is the least Herbrand model in [25, 21], and the set of atomic logical consequences in [18].

Abstract diagnosis is a generalization of declarative diagnosis, where we consider operational properties. An observable is any property which can be extracted from a goal computation, i.e., *observables are abstractions of SLD-trees*. Examples of useful observables are *computed answers*, *finite failures* and *call patterns* (i.e., procedure calls selected in an *SLD*-derivation). Other examples come from program analysis, e.g. *depth*(l)-*answers* (i.e., answers containing terms whose depth is $\leq l$), *types*, *modes* and *ground dependencies*. As we will discuss later, the relation among the observables can naturally be understood in terms of abstract interpretation.

Here are some motivations for abstract diagnosis.

- The most natural abstract diagnosis for positive logic programs is diagnosis w.r.t. computed answers, which leads to a more precise analysis, since declarative diagnosis is related to correct answers only.

- Diagnosis w.r.t. finite failures allows us to verify another relevant behavior, which has also a logical interpretation.

- Less abstract observables, such as call patterns, can be useful to verify the control and data flow between different procedures, as we usually do in interactive debugging. For example, the intended behavior that we specify might be a set of assertions of the form "the execution of the procedure call $p(t_1, \ldots, t_n)$ leads to the procedure call $q(s_1, \ldots, s_m)$".

- Diagnosis w.r.t. depth(l)-answers makes diagnosis w.r.t. computed answers effective, since both $\mathcal{I}_\alpha$ and $[\![P]\!]_\alpha$ are finite.

- Diagnosis w.r.t. types allows us to detect bugs as the *inadmissible calls* in [24]. If $\mathcal{I}_\alpha$ specifies the intended program behavior w.r.t. types, abstract diagnosis boils down to type checking.

- Diagnosis w.r.t. modes and ground dependencies allows us to verify other partial program properties.

In declarative diagnosis, the specification is usually assumed to be given by means of an *oracle*. This approach is feasible even in abstract diagnosis. However, since our method can handle abstractions, we can easily come out with finite observable behaviors and specify them in an extensional way.

The idea of combining abstract interpretation and debugging was first proposed in [5], where abstract interpretation techniques are used to statically determine the origin of bugs in higher-order imperative languages. The result is a set of correctness conditions expressed in terms of assertions.

Our theory of abstract diagnosis is built on an algebraic semantic framework for positive logic programs [7, 8], based on the formalization of observables as abstractions. A complete description of the framework is outside the scope of this paper. In Section 2 we summarize the main properties of the framework. We will consider here an important class of observables (*denotational observables*) with strong semantic properties. The diagnosis problem and the diagnosis algorithms for denotational observables introduced in [9] are considered in Section 3. We show that the existing declarative diagnosis methods can be reconstructed as instances of abstract diagnosis w.r.t. denotational observables. As in the case of declarative diagnosis, incorrect clauses can be detected by applying an immediate consequences operator to the specification. The first contribution of this paper is a method to detect incompleteness errors, which is similar to the incorrectness detection method, since it is based on the application of the (abstract) immediate consequences operator $T_{P,\alpha}$ to the specification. The main result is that this method is correct and complete, if $T_{P,\alpha}$ has a unique fixpoint. In Section 4 we show that this is the case for a large class of programs (acceptable programs). Acceptable programs were defined in [2] to study termination and all the pure PROLOG programs in [26] are reported to be acceptable. The same property is then shown (Section 5) to hold for all programs and for any observable α belonging to a suitable class of denotational observables. We finally consider in Section 6 the problem of diagnosis of incompleteness for a weaker class of observables (which are called *semi-denotational*). Semi-denotational observables are typically the properties used in program analysis.

2 Observables

We consider pure logic programs with the PROLOG (leftmost) selection rule. We assume the reader to be familiar with the notions of *SLD*-resolution and *SLD*-tree (see [22, 1]). The theory of observables [7, 8] is based on a *kernel semantics* for *SLD*-trees. The kernel semantics is given by two separate constructions, i.e.,

- a definition in denotational style,

- a definition given in terms of a transition system.

Both definitions are expressed in terms of three semantic operators $\otimes$, $\oplus$ and $\bowtie$, which are the semantic counterparts of the syntactic operators $\wedge$, $\vee$ and $\leftarrow$. The denotational and operational definitions are equivalent. Moreover, there exists a goal-independent program denotation which has the following properties:

- it can be defined in terms of the transition system (top-down definition $\mathcal{O}(P)$), by considering the set of *SLD*-trees for most general atomic goals.

- it can be obtained from the denotational definition (bottom-up definition $\mathcal{F}(P)$), by taking the least fixpoint of the operator T_P (the denotational semantics of P).
- $\mathcal{O}(P) = \mathcal{F}(P)$.
- the denotation is correct and minimal, i.e., $P_1 \approx P_2 \longleftrightarrow \mathcal{O}(P_1) = \mathcal{O}(P_2)$, where $\approx$ is the observational program equivalence induced by *SLD*-trees.
- the denotation is AND-compositional, i.e., we can derive the *SLD*-trees for any (conjunctive) goal from $\mathcal{O}(P)$.
- the denotation is OR-compositional, i.e., we can derive from $\mathcal{O}(P_1)$ and $\mathcal{O}(P_2)$ the denotation of $P_1 \bigcup P_2$.

Observables are abstractions of *SLD*-trees. More precisely, an *observable* is a function α from the domain of *SLD*-trees $\mathcal{R}$ to an abstract domain $\mathcal{D}$, which preserves the partial orders. α is an abstraction function according to abstract interpretation, i.e., there exists a function γ (concretization) from $\mathcal{D}$ to $\mathcal{R}$, such that (α, γ) is a Galois insertion. The theory of abstract interpretation tells us that we can define the most precise abstract version f^α of each semantic operator f as $f^\alpha = \alpha \circ f \circ \gamma$. Now we can obtain an abstract transition system and an abstract denotational definition from the ones of the kernel semantics, by simply replacing the operators $\otimes$, $\oplus$ and $\bowtie$ by their most precise abstract versions $\otimes_\alpha$, $\oplus_\alpha$ and $\bowtie_\alpha$. We obtain two abstract (goal-independent) program denotations: the top-down denotation $\mathcal{O}_\alpha(P)$ and the bottom-up denotation $\mathcal{F}_\alpha(P)$.

[8] gives a classification of observables, where each class is characterized by a set of simple axioms relating α, γ, $\otimes$, $\oplus$ and $\bowtie$.

- *perfect observables.* For perfect observables we can compute on the abstract domain, both operationally and denotationally, without losing precision. In particular, the abstract denotations are *precise*, i.e.,$\mathcal{O}_\alpha(P) = \mathcal{F}_\alpha(P) = \alpha(\mathcal{O}(P))$. Perfect observables have all the properties of the kernel semantics. Computed resultants is an example of a perfect observable.
- *denotational observables.* The abstract denotations are not precise. However, we can take the most precise approximation $T_{P,\alpha}$ of the T_P operator and use it in the denotational definition. The resulting abstract denotational semantics is now precise, as is the case for the bottom-up denotation $T_{P,\alpha} \uparrow \omega = \alpha(\mathcal{O}(P))$. Denotational observables have all the properties of the kernel semantics (restricted to the bottom-up denotations), apart from OR-compositionality. The abstract transition system cannot be made precise. Examples of denotational observables are: partial answers, call patterns, computed answers, correct answers, ground instances of computed answers. Some of the specialized bottom-up operators $T_{P,\alpha} = \alpha \circ T_P \circ \gamma$ are existing "immediate consequences operators". In particular,

  - If ϕ is the observable "ground instances of computed answers", $T_{P,\phi}$ is the ground operator defined in [27] (and $\mathcal{F}_\phi(P)$ is the least Herbrand model).
  - If ψ is the observable "correct answers", $T_{P,\psi}$ is the non-ground operator first defined in [6] (and $\mathcal{F}_\psi(P)$ is the least term model).
  - If ξ is the observable "computed answers", $T_{P,\xi}$ is the *s*-semantics operator defined in [16].
  - If η is the observable "call patterns", $T_{P,\eta}$ is the call patterns operator defined in [20].

  Perfect observables are also denotational.
- *operational observables.* These are observables for which we can systematically derive a precise abstract transition system, while the denotational semantics is not precise. This class is not relevant to our approach to diagnosis, which is based on $T_{P,\alpha}$.
- *semi-perfect observables.* The top-down and bottom-up denotations are equivalent, yet they are not precise, i.e., $\mathcal{O}_\alpha(P) = \mathcal{F}_\alpha(P) \succeq \alpha(\mathcal{O}(P))$, where $\preceq$ is the partial order relation on the abstract domain. Both the top-down and the bottom-up abstract computations are correct according to abstract interpretation theory, i.e., there is a loss of precision due to approximation. Semi-perfect observables have all the properties of the kernel semantics.
- *semi-denotational observables.* By taking the most precise approximation $T_{P,\alpha}$ of the T_P operator, we obtain a bottom-up abstract denotation which is more precise of the top-down abstract denotation, yet is less precise than the abstraction of the concrete denotation, i.e., $\alpha(\mathcal{O}(P)) \preceq T_{P,\alpha} \uparrow \omega \preceq \mathcal{O}_\alpha(P)$. Semi-denotational observables have the same properties of denotational observables, apart from the precision. Examples of semi-denotational observables are several domains used to abstract substitutions in the framework of program analysis (types, groundness dependencies, etc.). Semi-perfect observables are also semi-denotational.

Our basic theory of abstract diagnosis will be developed for denotational observables. In Section 6 we will mention how it can be extended to semi-denotational observables.

We show two of the $T_{P,\alpha}$ operators that will be later used in the examples.

- (computed answer substitutions)

$$
\begin{aligned}
T_{P,\xi}(I) = \{\, \langle p(\tilde{X}), \vartheta \rangle \mid \;& \\
& 1.\quad \tilde{X} \text{ is a tuple of new distinct variables} \\
& 2.\quad p(\tilde{t}) :\!-\, p_1(\tilde{t_1}), \ldots, p_n(\tilde{t_n}) \in P \\
& 3.\quad \langle p_i(\tilde{X}_i), \vartheta_i \rangle \in I,\ 1 \le i \le n, \\
& 4.\quad \vartheta = \mathsf{mgu}\,((p(\tilde{t}), p_1(\tilde{t_1}), \ldots, p_n(\tilde{t_n})), \\
& \qquad\qquad (p(\tilde{X}), p_1(\tilde{X}_1)\vartheta_1, \ldots, p_n(\tilde{X}_n)\vartheta_n))\,\}.
\end{aligned}
$$

- (l-answers with depth)

$$
\begin{array}{rll}
T_{P,\equiv}(I) = \{ \langle p(\tilde{X}), \vartheta, m\rangle \mid & & \\
& 1. & \tilde{X} \text{ is a tuple of new distinct variables} \\
& 2. & p(\tilde{t}) \text{ :- } p_1(\tilde{t_1}), \ldots, p_n(\tilde{t_n}) \in P \\
& 3. & \langle p_i(\tilde{X}_i), \vartheta_i, m_i\rangle \in I,\ 1 \leq i \leq n, \\
& 4. & \vartheta = \mathsf{mgu}\,((p(\tilde{t}), p_1(\tilde{t_1}), \ldots, p_n(\tilde{t_n})), \\
& & \qquad (p(\tilde{X}), p_1(\tilde{X}_1)\vartheta_1, \ldots, p_n(\tilde{X}_n)\vartheta_n)) \\
& 5. & m = 1 + m_1 + \ldots + m_n \leq l\,\}.
\end{array}
$$

3 Abstract diagnosis w.r.t. denotational observables

Let P be a program. If α is a denotational observable, we know that the actual and the intended behaviors of P for all the goals are uniquely determined by the behaviors for most general goals. The following Definitions 3.1 and 3.2 extend to abstract diagnosis the definitions given in [25, 18, 21] for declarative diagnosis. In the following $\mathcal{I}_\alpha$ is the specification of the abstraction of the intended behavior of program P for most general atomic goals w.r.t. the denotational observable α (i.e., $\mathcal{I}_\alpha$ is the specification of the intended $\alpha(\mathcal{O}(P))$). The actual abstract semantics of the program P is the abstract bottom-up denotation $\mathcal{F}_\alpha(P) = T_{P,\alpha} \uparrow \omega$, since α is a denotational observable. In the case of denotational observables we can assume the partial order on the abstract domain to be $\subseteq$ (set inclusion).

Definition 3.1

i. *P is* partially correct *w.r.t.* $\mathcal{I}_\alpha$, *if* $\mathcal{F}_\alpha(P) \subseteq \mathcal{I}_\alpha$.

ii. *P is* complete *w.r.t.* $\mathcal{I}_\alpha$, *if* $\mathcal{I}_\alpha \subseteq \mathcal{F}_\alpha(P)$.

iii. *P is* totally correct *w.r.t.* $\mathcal{I}_\alpha$, *if* $\mathcal{F}_\alpha(P) = \mathcal{I}_\alpha$.

If P is not totally correct, we are left with the problem of determining the errors, which are based on the *symptoms*.

Definition 3.2

i. *An* incorrectness symptom *is an element* σ *such that* $\sigma \in \mathcal{F}_\alpha(P)$ *and* $\sigma \notin \mathcal{I}_\alpha$.

ii. *An* incompleteness symptom *is an element* σ *such that* $\sigma \in \mathcal{I}_\alpha$ *and* $\sigma \notin \mathcal{F}_\alpha(P)$.

Note that a totally correct program has no incorrectness and no incompleteness symptoms. Our incompleteness symptoms are related to the insufficiency symptoms in [18], which are defined by taking $\mathsf{gfp}\,(T_P)$ instead of $\mathsf{lfp}\,(T_P)$ as program semantics. The two definitions, even if different, turn out to be the same for the

class of programs we are interested in (see the acceptable programs in Section 4). Ferrand's choice is motivated by the fact that $\mathsf{gfp}\,(T_P)$ is related to finite failures. The approach of using two different semantics for reasoning about incorrectness and incompleteness has been pursued in [19], leading to an elegant uniform (yet non-effective) characterization of correctness and completeness.

It is worth noting that we can reconstruct the usual definitions of declarative diagnosis within our more general framework, thus showing that the use of declarative specifications can also be motivated by operational arguments (i.e., the declarative semantics are goal-independent denotations corresponding to suitable denotational observables). In particular,

- the observable ϕ (ground instances of computed answers) gives us the declarative diagnosis based on the least Herbrand model [25, 21];

- the observable ψ (correct answers) gives us the declarative diagnosis based on the least term model [18].

It is straightforward to realize that an element may sometimes be an (incorrectness or incompleteness) symptom, just because of another symptom. The *diagnosis* determines the "basic" symptoms, and, in the case of incorrectness, the relevant clause in the program. This is captured by the definitions of *incorrect clause* and *uncovered element*, which are related to incorrectness and incompleteness symptoms, respectively.

Definition 3.3 *If there exists an element* σ *such that* $\sigma \notin \mathcal{I}_\alpha$ *and* $\sigma \in T_{\{c\},\alpha}(\mathcal{I}_\alpha)$, *then the clause* $c \in P$ *is* incorrect *on* σ.

Informally, c is incorrect on σ, if it derives a wrong observation from the intended semantics. $T_{\{c\},\alpha}$ is the operator associated to the program $\{c\}$, consisting of the clause c only.

The following theorem shows the relation between partial correctness (Definition 3.1) and absence of incorrect clauses (Definition 3.3). The theorem shows the feasibility of a diagnosis method for incorrectness based on the comparison between $\mathcal{I}_\alpha$ and $T_{P,\alpha}(\mathcal{I}_\alpha)$ and does not require to actually compute the denotation $\mathcal{F}_\alpha(P)$ (i.e., the least fixpoint of $T_{P,\alpha}$). Note that the second part of the theorem asserts that there might be incorrect clauses even if there are no incorrectness symptoms. In other words, if we just look at the semantics of the program, some incorrectness bugs can be "hidden" (because of an incompleteness bug).

Theorem 3.4 *If there are no incorrect clauses in P according to Definition 3.3, then P is partially correct w.r.t. α according to Definition 3.1 (hence there are no incorrectness symptoms). The converse does not hold.*

Proof.

i. If $T_{P,\alpha}(\mathcal{I}_\alpha) \subseteq \mathcal{I}_\alpha$, then $\mathcal{I}_\alpha$ is a pre-fixpoint of $T_{P,\alpha}$. Since $\mathcal{F}_\alpha(P) = \mathsf{lfp}\,(T_{P,\alpha})$ [7], by Tarski's theorem $\mathcal{F}_\alpha(P) \subseteq \mathcal{I}_\alpha$.

If $T_{P,\alpha}(\mathcal{I}_\alpha) \not\subseteq \mathcal{I}_\alpha$, then for some element σ, $\sigma \in T_{P,\alpha}(\mathcal{I}_\alpha)$ and $\sigma \notin \mathcal{I}_\alpha$. Hence, there exists a clause c in P such that $\sigma \in T_{\{c\},\alpha}(\mathcal{I}_\alpha)$. Therefore c is incorrect. Otherwise, if $T_{P,\alpha}(\mathcal{I}_\alpha) \subseteq \mathcal{I}_\alpha$ for all $c \in P$ and $\sigma \in T_{\{c\},\alpha}(\mathcal{I}_\alpha)$, then $\sigma \in T_{P,\alpha}(\mathcal{I}_\alpha)$. Hence $\sigma \in \mathcal{I}_\alpha$.

ii. Consider the program $P = \{\,\texttt{p :- r.}\,\}$ and the specification $\mathcal{I}_\xi = \{\,\langle r, \varepsilon\rangle\,\}$. P is partially correct because $\mathcal{F}_\xi(P) = \emptyset \subseteq \mathcal{I}_\xi$. However the only clause of P is incorrect because $\{\,\langle p, \varepsilon\rangle\,\} \in T_{P,\xi}(\mathcal{I}_\xi) - \mathcal{I}_\xi$.

■

As in the case of declarative diagnosis, handling completeness turns out to be more complex, since some incompletenesses cannot be detected by comparing $\mathcal{I}_\alpha$ and $T_{P,\alpha}(\mathcal{I}_\alpha)$. One would like to base the diagnosis on the following definition.

Definition 3.5 *An element σ is* uncovered *if*

$$\sigma \in \mathcal{I}_\alpha, \sigma \notin T_{P,\alpha}(\mathcal{I}_\alpha).$$

Informally, σ is uncovered if there are no clauses deriving it from the intended semantics.

The following proposition shows that we cannot base the diagnosis of incompleteness on the detection of uncovered elements.

Proposition 3.6 *There exist a program P, a denotational observable α and a specification $\mathcal{I}_\alpha$, such that*

i. there are no uncovered elements in P,

ii. P is not complete w.r.t. $\mathcal{I}_\alpha$ (i.e., there exist incompleteness symptoms).

Proof. Consider the program $P = \{\,\texttt{p(x) :- p(x).}\,\}$ and the specification $\mathcal{I}_\xi = \{\,\langle p(x), \varepsilon\rangle\,\}$. Then $T_{P,\xi}(\mathcal{I}_\xi) = \{\,\langle p(x), \varepsilon\rangle\,\}$, while $\mathcal{F}_\xi(P) = \emptyset$. ■

However, the following theorem shows that the diagnosis of incompleteness can be based on Definition 3.5 if the operator $T_{P,\alpha}$ has a unique fixpoint.

Theorem 3.7 *Assume $T_{P,\alpha}$ has a unique fixpoint. If there are no uncovered elements, then P is complete w.r.t. $\mathcal{I}_\alpha$ (hence there are no incompleteness symptoms). The converse does not hold.*

Proof.

i. If $\mathcal{I}_\alpha \subseteq T_{P,\alpha}(\mathcal{I}_\alpha)$, then $\mathcal{I}_\alpha$ is a post-fixpoint of $T_{P,\alpha}$. By Tarski's theorem, $\mathcal{I}_\alpha \subseteq \mathsf{gfp}\,(T_{P,\alpha})$. Since $\mathcal{F}_\alpha(P) = \mathsf{lfp}\,(T_{P,\alpha})$ [7] and $\mathsf{gfp}\,(T_{P,\alpha}) = \mathsf{lfp}\,(T_{P,\alpha})$, the thesis holds.

ii. Consider the program $P = \{\,\texttt{p :- r., r.}\,\}$ and the specification $\mathcal{I}_\xi = \{\,\langle p, \varepsilon\rangle\,\}$. P is complete because $\mathcal{F}_\xi(P) = \{\,\langle p, \varepsilon\rangle, \langle r, \varepsilon\rangle\,\} \supset \mathcal{I}_\xi$. However the element $\langle p, \varepsilon\rangle$ is uncovered because $T_{P,\xi}(\mathcal{I}_\xi) = \{\,\langle r, \varepsilon\rangle\,\}$.

■

In the next two sections we will consider two large classes of programs and denotational observables, for which $T_{P,\alpha}$ has a unique fixpoint. For these programs and observables, the diagnosis of incompleteness is as simple as the one for incorrectness. Note that, if $T_{P,\alpha}$ has a unique fixpoint, $\mathsf{lfp}\,(T_{P,\alpha}) = \mathsf{gfp}\,(T_{P,\alpha})$. Hence our incompleteness symptoms correspond to the insufficiency symptoms in [18].

The following corollary is a justification of the overall diagnosis method.

Corollary 3.8 *Assume $T_{P,\alpha}$ has a unique fixpoint. Then P is totally correct w.r.t. $\mathcal{I}_\alpha$, if and only if there are no incorrect clauses and uncovered elements according to definitions 3.3 and 3.5.*

If the abstraction α guarantees that for each most general atomic goal we have finitely many observations, then the specification is finite and our diagnosis is effective. In such a case, as already mentioned, $\mathcal{I}_\alpha$ can be specified in an extensional way and there is no need for the oracle.

4 Abstract diagnosis of acceptable programs

We consider here the abstract diagnosis of programs belonging to the class of *acceptable programs* [2], whose definition is given below. It is worth noting that acceptable programs are the left-terminating programs, i.e., those programs for which the *SLD*-derivations of ground goals (via the leftmost selection rule) are finite. As already mentioned, most interesting programs are acceptable (all the pure PROLOG programs in [26] are acceptable). The same property holds for most of the "wrong" versions of acceptable programs, since most "natural" errors do not affect the left-termination property.

Definition 4.1 [2] *A* level mapping *for a program P is a function $|\cdot| : B_P \longrightarrow \mathbb{N}$ from ground atoms to natural numbers. Let $|\cdot|$ be a level mapping for P and $\mathcal{I}$ be a (not necessarily Herbrand) model of P. P is* acceptable *w.r.t. $|\cdot|$ and $\mathcal{I}$, if for every clause a :- $b_1, \ldots b_n$ in $Ground(P)$ the following implication holds for $i \in [1, n]$:*

$$\mathcal{I} \models \wedge_{j=1}^{i-1} b_j \Longrightarrow |a| > |b_i|.$$

One relevant technical property of acceptable programs is that the ground immediate consequences operator has a unique fixpoint [2]. We have proved the following theorem, which tells us that the same property holds for all the operators $T_{P,\alpha}$, such that α is a denotational observable (*SLD*-trees, call patterns, answers with depth, *l*-answers with depth, correct and computed answers, ground instances of computed answers, etc.). We first need some additional definitions and lemmata. In the following, T_P denotes the immediate consequences operator of the kernel semantics.

Definition 4.2 *A* norm *for a program P on $\mathcal{R}$ is a function $\|\cdot\| : \mathcal{R} \to \mathbb{N}$ such that for every n the set $\{ x \in \mathcal{R} \mid \|x\| = n \}$ is finite. A program P is* $\mathcal{R}$-acceptable, *if there exists a norm s.t. for all $c \in P$ and all finite $I \in \mathcal{R}$*

$$\|T_{\{c\}}(I)\| > \|I\|.$$

Lemma 4.3 *Every acceptable program P is $\mathcal{R}$-acceptable.*

Proof. We just need to define $\|X\| = \max\{ |G\vartheta|^* \mid \langle G, \vartheta, \tilde{b}, cl\rangle \in X, \vartheta \}$, where $|\cdot|$ is the level mapping of P and $|B|^* = \min\{ |B\psi| \mid \psi \text{ is grounding for } B \}$. ■

For every observable α and each n we can define a "projection" function $\pi_{n,\alpha}(I) = \alpha \circ \pi_n \circ \gamma$, where $\pi_n(I) = \{ x \in I \mid \|x\| = n \}$. The functions $\pi_{n,\alpha}$ are well defined if α is a denotational observable.

Lemma 4.4 *Let P be an acceptable program and α be a denotational observable. Then*

$$\pi_{n,\alpha} \circ T_{P,\alpha} = \pi_{n,\alpha} \circ T_{P,\alpha} \circ (\sum_{i<n} \pi_{i,\alpha}).$$

Proof. For every n, every I and for all $m \geq n$, the sets $(\pi_n \circ T_P \circ \pi_m)(I)$ are empty, because $\|T_P(I)\| > \|I\|$ by hypothesis . Thus $\pi_n \circ T_P = \pi_n \circ T_P \circ \sum_{i<n} \pi_i$. Then for every α

$$\begin{aligned}
\pi_{n,\alpha} \circ T_{P,\alpha} &= \alpha \circ \pi_n \circ \gamma \circ \alpha \circ T_P \circ \gamma \\
&= \alpha \circ \pi_n \circ T_P \circ \gamma \\
&= \alpha \circ (\pi_n \circ T_P \circ \sum_{i<n} \pi_i) \circ \gamma \\
&= \alpha \circ \pi_n \circ \gamma \circ \alpha \circ T_P \circ \gamma \circ \alpha \circ \sum_{i<n} (\pi_i \circ \gamma) \\
&= (\alpha \circ \pi_n \circ \gamma) \circ (\alpha \circ T_P \circ \gamma) \circ \sum_{i<n} (\alpha \circ \pi_i \circ \gamma) \\
&= \pi_{n,\alpha} \circ T_{P,\alpha} \circ \sum_{i<n} \pi_{i,\alpha}
\end{aligned}$$

which is the claim. ■

We are now ready to prove the main theorem.

Theorem 4.5 (fixpoint uniqueness) *Let P be an acceptable program and α be a denotational observable. Then $T_{P,\alpha}{\uparrow}\omega$ is the unique fixpoint of $T_{P,\alpha}$.*

Proof. Clearly $T_{P,\alpha}{\uparrow}\omega$ is a fixpoint. Now assume that X and Y are fixpoints. We show, by induction on n, that for all n, $\pi_{n,\alpha}(X) = \pi_{n,\alpha}(Y)$.

$$\begin{aligned}
\pi_{0,\alpha}(X) &= (\pi_{0,\alpha} \circ T_{P,\alpha})(X) \\
&= (\pi_{0,\alpha} \circ T_{P,\alpha})(\perp) \\
&= (\pi_{0,\alpha} \circ T_{P,\alpha})(Y) \\
&= \pi_{0,\alpha}(Y)
\end{aligned}$$

Moreover, if for all $i < n$ $\pi_{i,\alpha}(X) = \pi_{i,\alpha}(Y)$, then

$$\begin{aligned}
\pi_{n,\alpha}(X) &= (\pi_{n,\alpha} \circ T_{P,\alpha})(X) \\
&= (\pi_{n,\alpha} \circ T_{P,\alpha} \circ (\sum_{i<n} \pi_{i,\alpha}))(X) \\
&= (\pi_{n,\alpha} \circ T_{P,\alpha} \circ (\sum_{i<n} \pi_{i,\alpha}))(Y) \\
&= (\pi_{n,\alpha} \circ T_{P,\alpha})(Y) \\
&= \pi_{n,\alpha}(Y)
\end{aligned}$$

Hence

$$X = (\sum_{n \in \mathbb{N}} \pi_{n,\alpha})(X) = (\sum_{n \in \mathbb{N}} \pi_{n,\alpha})(Y) = Y.$$

■

Theorems 4.5 and 3.7 allow us to perform the diagnosis of incompleteness errors according to Definition 3.5.

Corollary 4.6 *Let P be an acceptable program. Then P is* totally correct *w.r.t. $\mathcal{I}_\alpha$ if and only if $T_{P,\alpha}(\mathcal{I}_\alpha) = \mathcal{I}_\alpha$.*

It is worth noting that the property of being acceptable is undecidable. Therefore we do not mean the diagnosis to contain a test for acceptability. We just want to remark that, since all sensible programs turn out to be acceptable, the diagnosis algorithm based on the application of the bottom-up operator to the specification (both for correctness and incompleteness) is indeed feasible.

Note that this result applies to declarative diagnosis as well, because, as we have shown in Section 3, it can be explained in terms of denotational observables.

Example 4.7 Consider the acceptable program P of Figure 1, which is an "ancestor" database with a missing clause (`ancestor(X, Y) :– parent(X, Y).`). Consider the computed answer substitutions observable ξ. The specification is

$$\begin{aligned}
\mathcal{I}_\xi = \{ &\langle parent(X,Y), \{ X/\text{terach}, Y/\text{abraham} \}\rangle, \\
&\langle parent(X,Y), \{ X/\text{abraham}, Y/\text{isaac} \}\rangle, \\
&\langle ancestor(X,Y), \{ X/\text{terach}, Y/\text{abraham} \}\rangle, \\
&\langle ancestor(X,Y), \{ X/\text{terach}, Y/\text{isaac} \}\rangle, \\
&\langle ancestor(X,Y), \{ X/\text{abraham}, Y/\text{isaac} \}\rangle \},
\end{aligned}$$

```
ancestor(X,Y) :- ancestor(X,Z),parent(Z,Y).
parent(abraham,isaac).
parent(terach,abraham).
```

Figure 1: A wrong acceptable program

while

$$\begin{aligned}T_{P,\xi}(\mathcal{I}_\xi) = \{\, &\langle parent(X,Y), \{\, X/\text{terach}, Y/\text{abraham}\,\}\rangle,\\ &\langle parent(X,Y), \{\, X/\text{abraham}, Y/\text{isaac}\,\}\rangle,\\ &\langle ancestor(X,Y), \{\, X/\text{terach}, Y/\text{isaac}\,\}\rangle\,\}.\end{aligned}$$

The elements $\langle ancestor(X,Y), \{\, X/\text{terach}, Y/\text{abraham}\,\}\rangle$ and $\langle ancestor(X,Y), \{\, X/\text{abraham}, Y/\text{isaac}\,\}\rangle$ are diagnosed as uncovered, while the "derived" element $\langle ancestor(X,Y), \{\, X/\text{terach}, Y/\text{isaac}\,\}\rangle$ is not, even if it is an incompleteness symptom. ■

5 Abstract diagnosis of acceptable denotational observables

In this section we show that Definition 3.5 can be used to detect incompleteness errors even for non acceptable programs, if the observable α satisfies a property which guarantees that the immediate consequences operator has a unique fixpoint (acceptable observables).

Definition 5.1 *An* α-level mapping *for a denotational observable* $\alpha : \mathcal{R} \to \mathcal{D}$ *is a function* $|\cdot| : \mathcal{D} \to \mathbb{N}$*. Let* $|\cdot|$ *be an* α*-level mapping,* α *is* acceptable *w.r.t.* $|\cdot|$ *if for every clause c and for all finite* $\mathcal{I}_\alpha$*,*

$$|T_{\{c\},\alpha}(\mathcal{I}_\alpha)| > |\mathcal{I}_\alpha|.$$

For every α-level mapping $|\cdot|$ we can define the norm $\|X\| = |\alpha(X)|$ and, therefore, the projections $\pi_{n,\alpha}$.

Lemma 5.2 *Let P be a program and* α *be an acceptable denotational observable. Then*

$$\pi_{n,\alpha} \circ T_{P,\alpha} = \pi_{n,\alpha} \circ T_{P,\alpha} \circ (\sum_{i<n} \pi_{i,\alpha}).$$

Theorem 5.3 *Let P be a program and* α *be an acceptable denotational observable. Then* $T_{P,\alpha}$ *has a unique fixpoint.*

We show now that the basic *SLD*-trees observable (*id*) is indeed acceptable. The abstraction can destroy this property. However all the denotational observables

which keep some information about the length of the derivation are also acceptable. In particular this is the case of the l-answers with depth observable, which has been proposed to achieve finite extensional specifications. On the other hand, correct and computed answers substitutions are not acceptable (as shown by the program in the proof of Proposition 3.6).

Proposition 5.4 *The identical denotational observable* $id : \mathcal{R} \longrightarrow \mathcal{R}$ *is acceptable.*

Proposition 5.5 *The observable* Ξ *is acceptable.*

Example 5.6 Consider the program P of Figure 2 which is another "wrong" version of the "ancestor" database. This version, however, is not acceptable (the computation of the goal `?- ancestor(terach,abraham)` goes into an infinite loop). We will show that the bug can be located by an acceptable denotational observable. Consider first the computed answer substitutions observable ξ, which is not acceptable.

$$\begin{aligned}\mathcal{I}_\xi = \{\, &\langle parent(X,Y), \{\, X/\text{terach}, Y/\text{abraham}\,\}\rangle,\\ &\langle parent(X,Y), \{\, X/\text{abraham}, Y/\text{isaac}\,\}\rangle,\\ &\langle ancestor(X,Y), \{\, X/\text{terach}, Y/\text{abraham}\,\}\rangle,\\ &\langle ancestor(X,Y), \{\, X/\text{terach}, Y/\text{isaac}\,\}\rangle,\\ &\langle ancestor(X,Y), \{\, X/\text{abraham}, Y/\text{isaac}\,\}\rangle\,\}.\end{aligned}$$

Even if $T_{P,\xi}(\mathcal{I}_\xi) = \mathcal{I}_\xi$, the program has an incompleteness symptom, since the element $\langle ancestor(X,Y), \{\, X/terach, Y/isaac\}\rangle$ does not belong to $\mathcal{F}_\xi(P)$.

Consider now the 6-answers with depth observable Ξ, which is instead acceptable.

$$\begin{aligned}\mathcal{I}_\Xi = \{\, &\langle parent(X,Y), \{\, X/\text{terach}, Y/\text{abraham}\,\}, 1\rangle,\\ &\langle parent(X,Y), \{\, X/\text{abraham}, Y/\text{isaac}\,\}, 1\rangle,\\ &\langle ancestor(X,Y), \{\, X/\text{terach}, Y/\text{abraham}\,\}, 2\rangle,\\ &\langle ancestor(X,Y), \{\, X/\text{terach}, Y/\text{isaac}\,\}, 4\rangle,\\ &\langle ancestor(X,Y), \{\, X/\text{abraham}, Y/\text{isaac}\,\}, 2\rangle\,\},\end{aligned}$$

$$\begin{aligned}T_{P,\Xi}(\mathcal{I}_\Xi) = \{\, &\langle parent(X,Y), \{\, X/\text{terach}, Y/\text{abraham}\,\}, 1\rangle,\\ &\langle parent(X,Y), \{\, X/\text{abraham}, Y/\text{isaac}\,\}, 1\rangle,\\ &\langle ancestor(X,Y), \{\, X/\text{terach}, Y/\text{abraham}\,\}, 2\rangle,\\ &\langle ancestor(X,Y), \{\, X/\text{terach}, Y/\text{isaac}\,\}, 6\rangle,\\ &\langle ancestor(X,Y), \{\, X/\text{abraham}, Y/\text{isaac}\,\}, 2\rangle\,\}.\end{aligned}$$

The diagnosis now detects the incorrect clause c_2 in addition to the uncovered element $\langle ancestor(X,Y), \{\, X/\text{terach}, Y/\text{isaac}\,\}, 4\rangle$. ■

6 Diagnosis of incompleteness for semi-denotational observables

```
c1) ancestor(X,Y) :- parent(X,Y).
c2) ancestor(X,Y) :- ancestor(X,Y), parent(Z,Y).
c3) parent(abraham, isaac).
c4) parent(terach, abraham).
```

Figure 2: A non acceptable program

Semi-denotational observables are meant to model the abstraction (with approximation) involved in program analysis (e.g. depth(l)-answers, types, modes, ground dependencies and sharing). Even the most precise abstract denotation $\mathcal{F}_\alpha(P)$ is just an approximation of the abstraction of the concrete semantics. Namely $\alpha(\mathcal{O}(P)) = \alpha(\mathcal{F}(P)) \preceq \mathcal{F}_\alpha(P)$. The specification $\mathcal{I}_\alpha$ is a specification of the intended behavior $\alpha(\mathcal{O}(P))$. Hence we cannot get any information about partial correctness, since in general the following relation holds (for a complete program):

$$\mathcal{I}_\alpha \preceq \mathcal{F}_\alpha(P).$$

In other words, in a partially correct and complete program, the actual program denotation and the specification can be different, just because of the approximation introduced by the semi-denotational observable.

On the other hand, the definitions given in Section 3 related to completeness (and the corresponding diagnosis algorithm for detecting uncovered elements) are applicable to the case of semi-denotational observables as well, once we adapt our definitions to the partial order $\preceq$, which is usually different from set inclusion in semi-denotational observables. In particular, we can decide completeness by comparing $\mathcal{I}_\alpha$ and $T_{P,\alpha}(\mathcal{I}_\alpha)$, if P is acceptable and α is a semi-denotational observable.

7 Conclusions

We have shown that the theory of declarative diagnosis can be extended to the case where the specification defines the intended behavior of programs w.r.t. operational properties which can be formalized as denotational or semi-denotational observables, as first suggested in [9]. The main new result w.r.t. [9] is the simple characterization of incompleteness in the case of acceptable programs or acceptable observables.

This paper is concerned with the foundations of abstract diagnosis. Hence we have not dealt with the problems of designing efficient diagnosis algorithms and of implementing the specification. Let us just mention that we can easily define top-down diagnosis algorithms, in the style of those discussed in [23], where the specification is given by an oracle, possibly implemented by querying the user. One such an algorithm, for the case of the computed answers denotational observable, is described in [10]. The top-down diagnoser uses one oracle only, and does not

require to determine the symptoms in advance. The AND-compositionality property of $\mathcal{F}_\alpha(P)$ allows us to determine all the incorrect clauses and uncovered elements by considering just a finite set of atomic goals (i.e., the most general atomic goals).

The effectivity of the diagnosers relies on our ability to handle finite approximations of the specification. In fact, if $\mathcal{I}_\alpha$ is not finite, the diagnosis is unfeasible since the oracle may return infinite answers to some queries. Abstract diagnosis allows us to tackle this problem, by considering abstractions (modeled by denotational observables) on finite domains. One example is the observable l-answers with depth considered in this paper, which, however, requires the user to reason in unacceptable operational terms. A second solution is to move to more natural observables, such as depth(l)-answers, which can be modeled as a semi-denotational observable (in this case, however, we can only reason about incompleteness). Finally, we can resort to *partial specifications* as defined in [10] in the case of the computed answers observable. Partial specifications are simply descriptions of finite approximations of the intended program behavior. The theory of abstract diagnosis can be extended to partial specifications, resulting in weaker results, which may be, however, very useful in the practice of diagnosis.

References

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

[2] K. R. Apt and D. Pedreschi. Reasoning about termination of pure PROLOG programs. *Information and Computation*, 106(1):109–157, 1993.

[3] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19-20:149–197, 1994.

[4] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.

[5] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation '93*, pages 46–55, 1993.

[6] K. L. Clark. Predicate logic as a computational formalism. Res. Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979.

[7] M. Comini and G. Levi. An algebraic theory of observables. In M. Bruynooghe, editor, *Proceedings of the 1994 Int'l Symposium on Logic Programming*, pages 172–186. The MIT Press, Cambridge, Mass., 1994.

[8] M. Comini, G. Levi, and M. C. Meo. Compositionality of *SLD*-derivations and their abstractions. In M.I. Sessa, editor, *Proceedings GULP-PRODE'95*, 1995.

[9] M. Comini, G. Levi, and G. Vitiello. Abstract debugging of logic programs. In L. Fribourg and F. Turini, editors, *Proc. Logic Program Synthesis and Transformation and Metaprogramming in Logic 1994*, volume 883 of *Lecture Notes in Computer Science*, pages 440–450. Springer-Verlag, Berlin, 1994.

[10] M. Comini, G. Levi, and G. Vitiello. Declarative diagnosis revisited. In M.I. Sessa, editor, *Proceedings GULP-PRODE'95*, 1995.

[11] M. Comini, G. Levi, and G. Vitiello. Efficent detection of incompleteness errors in the abstract debugging of logic programs. In M. Ducassé, editor, *Proc. 2nd International Workshop on Automated and Algoritmic Debugging, AADEBUG'95*, 1995.

[12] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

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

[14] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[15] M. Ducassè and J. Noyè. Logic programming environments: Dynamic program analysis and debugging. *Journal of Logic Programming*, 19-20:351–384, 1994.

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

[17] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 102(1):86–113, 1993.

[18] G. Ferrand. Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Method. *Journal of Logic Programming*, 4:177–198, 1987.

[19] G. Ferrand. The notions of symptom and error in declarative diagnosis of logic programs. In P. A. Fritzson, editor, *Automated and Algorithmic Debugging, Proc. AADEBUG '93*, volume 749 of *Lecture Notes in Computer Science*, pages 40–57. Springer-Verlag, Berlin, 1993.

[20] M. Gabbrielli, G. Levi, and M. C. Meo. Observational Equivalences for Logic Programs. In K. Apt, editor, *Proc. Joint Int'l Conf. and Symposium on Logic Programming*, pages 131–145. The MIT Press, Cambridge, Mass., 1992. Extended version to appear in *Information and Computation*.

[21] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

[22] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[23] L. Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10:255–285, 1991.

[24] L. M. Pereira. Rational debugging in logic programming. In E. Y. Shapiro, editor, *Proceedings of the 3rd International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 203–210. Springer-Verlag, Berlin, 1986.

[25] E. Y. Shapiro. Algorithmic program debugging. In *Proc. Ninth Annual ACM Symp. on Principles of Programming Languages*, pages 412–531. ACM Press, 1982.

[26] L. Sterling and E. Y. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., 1986.

[27] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.