Una fruttuosa esperienza in Logica Computazionale
A valuable experience in Computational Logic

Annalisa Bossi, Nicoletta Cocco
Dipartimento di Informatica, Università Ca' Foscari di Venezia,
via Torino 155, 30172, Venezia, Italy
email: {bossi, cocco}@dsi.unive.it

SOMMARIO/*ABSTRACT*

Illustriamo qui brevemente la nostra esperienza nel campo della verifica e delle trasformazioni dei programmi logici. Pur occupandoci ora di tematiche completamente diverse, verifica di proprietà di sicurezza da un lato e analisi di sistemi biologici dall'altro, continuiamo ad utilizzare proficuamente la nostra precedente esperienza.

In this paper, we briefly describe our esperience in the field of verification and transformation of logic programming. Though now we are working in a completely different field, verification of security properties on one hand and biosystems analysis on the other, our previous experience continues to be a valuable guide.

Keywords: logic programming, termination verification, program transformation

1 Introduction

It is very pleasant to remember the time we spent in working on Logic Programming. The friendship and warmth of the people we met, the enthusiasm and interest in research, the curiosity and joy of young researchers, have been strong reasons for working in this field and to be happy with it.

Our main interest, since the beginning, was analysis and verification of logic programs and program transformation. After more than fifteen years of happy and satisfactory research, we felt the need to enlarge our research field and we tried to export our expertise in Computational Logic to different research topics.

In the following section we briefly resume our main results in the field of logic programming and then we give a brief account of our present research interests.

2 Our contribution to Logic Programming

Programming methodology imposes to focus first on the correctness of a program and only later on its efficiency. This is necessary also in logic programming and it requires both program verification and program optimization tools. Our research in LP has been motivated by these needs, dealing mainly with *transformation systems* and with *analysis techniques*.

Analysis techniques.
Logic programs are declarative in essence, and this is a great advantage for programs prototyping and development. Nevertheless, there are properties which are not directly expressed by the program itself and have to be proved. We proposed a technique for verifying correctness and completeness of a logic program with respect to a Pre/Post declarative specification of data properties [1]. This can be used to guarantee both the correspondence of the program to its intended meaning and the applicability of program transformations. We considered also the operational property on having successes, or finite failures, which is relevant for query correctness and efficiency [5, 6]. Besides, the property of not having finite failures can be used to simplify applicability conditions of program transformation operations.

Techniques for verifying termination.
Termination is an essential property of programs. We considered the problem of verifying *universal termination* of logic programs. This is a rather strong requirement for a query, namely to have only finite LD-derivations[1]. All the methods to solve this problem, if effective, can only provide sufficient criteria for termination. In our works we developed various methods for the analysis of universal termination by considering different classes of programs which can be verified.

We introduced a class of functions to weight the terms occurring in a program (*semilinear norms*) [16, 18]. The norms in this class provide a syntactical characterization

[1] SLD-derivations build with the leftmost selection rule.

of rigid terms, i.e. terms whose weight does not change under substitution. The notion of rigid term generalizes the notion of ground term. We defined a proof method for universal termination, based on Pre/Post conditions which deal with the rigidity of terms and can be derived by the mode and type properties of atoms. In [17] we generalized our previous work by considering also terms with a specified structure by means of *typed norms*. Besides, we studied how mode and type information can be used for characterizing termination properties. We defined the class of *well-moded programs* [31], namely programs which are inductively "well-formed" with respect to a specified input-output functionality. This allowed us to define and characterize *well-terminating programs*, namely programs for which all well-moded queries have only finite LD-derivations. We proposed also a termination property for general logic programs (programs with negation) [19]. A general program is *typed-terminating* if it terminates for any well-typed query. These definitions lead to sufficient conditions for termination which are compositional and simple to verify.

In [14] we completed our work on the verification of termination properties, by proposing a modular proof technique applicable to hierarchical general programs. Besides, by using mode or type information, it is possible to verify termination incrementally.

Trasformations on logic and Prolog programs.
Program transformations are applied both in program synthesis and in program optimization. For logic programs the "logic" component makes transformations very natural and easy to be studied formally. But, when we move to Prolog programs, non-declarative properties, like termination, cannot be ignored.

At first we focused on *program specialization*, which consists in restricting the applicability of the original program while optimizing it: the specialized program deals with fewer cases but in a more efficient way. Some parts of the computation become redundant, other parts can be pre-computed (partial evaluation). Specialization seems to fit very well logic programs in order to pass from a relational definition to some specific functionalities. We proposed a methodology for specializing a logic program [7] and studied a set of basic transformation operations which allow one (i) to associate a new application domain to the query by means of constraints, and (ii) to propagate them through the program for optimizing it. The set of basic operations includes:

- *new definition*, it defines a new predicate in terms of other predicates already available in the program;
- *unfold*, it substitutes an atom in a clause body with all its definitions;
- *fold*, it substitutes a set of atoms in a clause body with an equivalent atom;
- *prune*, it removes a redundant clause from the program;
- *thin*, it removes a redundant literal from a clause body;
- *fatten*, it adds further literals in a clause body whenever this allows for simplifications;
- *replace*, it substitutes a set of literals in a clause body for another set of literals; it is a generalization of the *thin* and *fatten* operations.

Each operation must produce a program which is equivalent to the original one, but more efficient. Program equivalence depends on the semantics we consider. Hence, we studied these transformation operations with respect to different program semantics. Our effort has been to determine sufficient conditions, simple to verify, for the various operations and semantics. We considered the classic semantics given by the minimal Herbrand model [7] and the semantics given by computed answers substitutions [2]. Moreover, in [8, 9] we considered general programs (with negation) and some semantics for them, such as Fitting's semantics, Kunen's semantics, and the Well-founded semantics.

Besides basic transformation operations, we defined *simultaneous replacement* and we studies it with respect to the three-valued completion of a logic program [11].

Any transformation system is a source-to-source rewriting methodology devised to improve the efficiency of a program. Any such transformation should preserve the main properties of the initial program. The transformation operations defined for logic programs do not consider operational properties, among them, termination. These properties become relevant for Prolog programs. To deal with that we followed two approaches.

On one hand, we considered *acyclic programs*, namely programs which terminate for each ground query and any selection rule, and *acceptable programs*, namely programs which terminate for each ground query and leftmost selection rule. For both of them we identified the subclasses of programs closed under unfold and fold operations [20, 11].

In order to be applicable most of the transformations require to reorder the atoms in clause bodies, then in [12] we extended the previous work by considering also a *switch* operation which allows one to reorder consecutive atoms.

On the other hand, in [3, 4] we followed a more operational approach and we defined a *non-increasing* property for a transformation. It is a very strong property which guarantees that the transformation is both preserving universal termination and optimizing, since it cannot increase the depth of the derivation tree associated to a query.

In [13] we considered and analyzed the main systems for transforming logic and Prolog programs. In particular we discuss if they preserve non-declarative properties of the original program and specifically termination properties.

Semantics for logic programs.
Our work on the semantics of logic programming is ruled by the convincement that a semantics should help in understanding the meaning of programs by providing useful notions of observable program equivalences. The *s-semantic approach* (see [26]) provides a methodology to define semantics which enjoy this property. Each semantics in the approach captures some observable properties

of logic programs and allows us to detect when two programs are undistinguishable by observing their behaviors, thus providing a suitable base for program analysis and transformation. Following this approach, we defined the Ω- *semantics*, a compositional semantics for positive logic programs. It provides a refined notion of observational equivalence which takes into account both computed answers and program composition by union of clauses [27].

Most logic programming languages offer some kind of *dynamic scheduling* to increase the expressive power and to control execution. But the presence of dynamic scheduling makes more complex the programs behaviour and more difficult the description of the semantics. *Input consuming* derivations have been introduced and studied in [21, 22, 23] to describe dynamic scheduling while abstracting from the technical details. In [15] we reviewed and compared the different proposals given for dynamic scheduling and in particular for the denotational semantics of programs with input consuming derivations. We also show how they can be applied to termination analysis.

3 Present Research

Verification of security properties.
In the recent years, security has gained more and more importance. In this field, our research focus on *information flow* properties, i.e., security properties that allow one to express constraints on how information should flow among different groups of entities. An interesting feature of these kind of properties, is that they protect the system even against internal attacks performed by, e.g., viruses or Trojan horses.

We study different classes of security properties and conditions to ensure global properties by means of local *unwinding conditions* [25]. Locality allows us to define a proof system which provides a very efficient technique for the development and verification of secure processes [24].

For many practical applications the requirement of a complete absence of any information flow could be stronger than necessary when some knowledge about the environment (context) in which the process is going to run is available. To relax this requirement we introduce a general notion of *secure contexts* for a process [28]. In [29] we moved from a process algebra setting to a standard programming environment. We present a general unwinding framework for the definition of information flow security properties of concurrent programs, described in a standard imperative language, admitting parallel executions on a shared memory.

Biosystems analysis.
Computational biology is a recent field combining computer science and molecular biology to study living beings. We focus our attention on two areas, pattern discovery and system biology.

Pattern discovery. Many biological problems require to blindly search into DNA or protein sequences for relevant signals. Often we may assume that strings which appears "strangely often" or "strangely rarely" in such sequences have an associated functional purpose. We studied the techniques for finding such signals and for giving them a compact representation as patterns. In particular we define *maximal patterns* [30], which correspond to the largest subsets of strings which can be grouped together. The set of maximal patterns is unique and very readable, intuitively it represents all possible "most abstract views" of the strings we are interested in. We propose two different algorithms for computing the set of maximal patterns.

Systems biology is a rather new field studying complex interactions in biological systems. The aim is to model such systems, to formally analyze their properties and to simulate their behaviour. This would make possible to do *in silico* experiments instead of *in vivo* experiments, which may be difficult, or even impossible, to perform on biological systems. Computational logic and formal techniques to specify and analyze concurrent processes can be applied to this field.

4 Acknowledgements

Our thanks to Matteo Baldoni and Cristina Baroglio for organizing this collection in honour of Alberto Martelli, a dear friend and one of the most attracting personality for the Italian Computational Logic community.

REFERENCES

[1] A. Bossi and N. Cocco. Verifying Correctness of Logic Programs. In J. Diaz and F. Orejas, editors, *Proceedings TAPSOFT '89*, Barcelona, Spain, LNCS 352, pp. 96–110, Springer-Verlag, 1989.

[2] A. Bossi, e N. Cocco. Basic Transformation Operations which preserve Computed Answer Substitutions of Logic Programs. *Journal of Logic Programming*, 16:47–87, 1993.

[3] A. Bossi and N. Cocco. Preserving universal termination through unfold/fold. In G. Levi and M. Rodríguez-Artalejo, editors, *Proceedings ALP'94*, LNCS 850, pp. 269–286, Springer-Verlag, 1994.

[4] A. Bossi and N. Cocco. Replacement Can Preserve Termination. In J. Gallagher, editor, *Proceedings LOPSTR'96*, LNCS 1207, pp.104–129, Springer-Verlag, 1997.

[5] A. Bossi and N. Cocco. Programs without Failures. In N. Fuchs, editor, *Proceedings LOPSTR'97*, LNCS 1463, pp. 28–48, Springer-Verlag,1998.

[6] A. Bossi and N. Cocco. Successes in Logic Programs. In P. Flener, editor, *Proceedings LOPSTR'98*, LNCS 1559, pp. 219–239, Springer-Verlag, 1999.

[7] A. Bossi, N. Cocco, e S. Dulli. A Method for Specializing Logic Programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, 1990.

[8] A. Bossi, N. Cocco, e S. Etalle. On Safe Folding. In M. Bruynooghe and M. Wirsing, editors, *Proceedings PLILP'92*, Leuven, Belgium, LNCS 631, pp. 172–186, Springer-Verlag, 1992.

[9] A. Bossi, N. Cocco, e S. Etalle. Transforming Normal Programs by Replacement. In A. Pettorossi, editor, *Proceedings META'92*, Uppsala, Sweden, LNCS 649, pp. 265–279, Springer-Verlag, 1992.

[10] A. Bossi, N. Cocco, and S. Etalle. Transformation of Left Terminating Programs: the Reordering Problem. In M. Proietti, editor, *Proceedings LOPSTR'95*, LNCS 1048, pp. 33–45, Springer-Verlag, 1995.

[11] A. Bossi, N. Cocco, e S. Etalle. Simultaneous Replacement in Normal Programs. *Journal of Logic and Computation*, 6(1):79–120, 1996.

[12] A. Bossi, N. Cocco, e S. Etalle. Transformation of Left Terminating Programs. In A. Bossi editor, *Proceedings of LOPSTR'99*, Venezia, Italy, LNCS 1817, pp. 156-175, Springer-Verlag, 2000.

[13] A. Bossi, N. Cocco e S. Etalle. Transformation Systems and Nondeclarative Properties. In A. Kakas and F. Sadri editors, *Computational Logic: Logic Programming and Beyond (Essays in honour of Robert A. Kowalski)*. LNAI 2407, pp. 162-186, Springer-Verlag, 2002.

[14] A. Bossi, S. Etalle N. Cocco, and S. Rossi. On Modular Termination Proofs of General Logic Programs. *Theory and Practice of Logic Programming*, 2(3):263–291, 2002.

[15] A. Bossi, S. Etalle N. Cocco, and S. Rossi. Declarative Semantics of Input-Consuming Logic Programs. In M. Bruynooghe, K. Lau editors, *Program Development in Computational Logic - A Decade of Research Advances in Logic-Based Program Development*. LNCS 3049, Springer-Verlag, 2004.

[16] A. Bossi, N. Cocco, and M. Fabris. Proving termination of logic programs by exploiting term properties. In S. Abramsky and T.S.E. Maibaum, editors, *Proceedings CCPSD-TAPSOFT'91*, LNCS 494, pp. 153–180, Springer-Verlag, 1991.

[17] A. Bossi, N. Cocco, and M. Fabris. Typed Norms. In Krieg-Bruckner, editor, *Proceedings ESOP'92*, Rennes, France, LNCS 582, pp. 73–92. Springer-Verlag, 1992.

[18] A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science*, 124:297–328, 1994.

[19] A. Bossi, N. Cocco, and S. Rossi. Termination of Well-Typed Logic Programs. In H. Sondergaard editor, *Proceedings PPDP'01*, Firenze, Italy, pp.73-81, ACM Press, 2001.

[20] A. Bossi and S. Etalle. Transforming Acyclic Programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1081–1096, July 1994.

[21] A. Bossi, S. Etalle, and S. Rossi. Semantics of well-moded input-consuming logic programs. *Computer Languages*, 26(1):1–25, 2000.

[22] A. Bossi, S. Etalle, and S. Rossi. Properties of input-consuming derivations. *Theory and Practice of Logic Programming*, 2(2):125–154, 2002.

[23] A. Bossi, S. Etalle, S. Rossi, and J.-G. Smaus. Termination of simply-moded logic programs with dynamic scheduling. *ACM Transactions on Computational Logic (TOCL)*, 5(3):470–507, 2004.

[24] A. Bossi, R. Focardi, C. Piazza, and S. Rossi. A proof system for information flow security. M. Leuschel, editor, *Proceedings LOPSTR'02*, LNCS 2664, pp. 2199–218, Springer-Verlag, 2003.

[25] A. Bossi, R. Focardi, C. Piazza, and S. Rossi. Verifying persistent security properties. *Computer Languages, Systems and Structures*, 30(3-4):231–258, 2004.

[26] A. Bossi, M. Gabrielli, G. Levi, and M. Martelli. The S-semantics approach: Theory and applications. *The Journal of Logic Programming*, 19 & 20:149–198, May 1994.

[27] 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.

[28] A. Bossi, D. Macedonio, C. Piazza, and S. Rossi. Information flow in secure contexts. *Journal of Computer Security*, 13(3):391–422, 2005.

[29] A. Bossi, C. Piazza, and S. Rossi. Compositional information flow security for concurrent programs. *Journal of Computer Security*, 15(3):373–413, 2007.

[30] N. Cocco and M. Simeoni, Maximal abstraction of strings. *Dipartimento di Informatica, Università Ca' Foscari di Venezia, Rapporto di ricerca* CS-2007-2, 2007.

[31] S. Etalle, A. Bossi, and N. Cocco. Termination of well-moded programs. *Journal of Logic Programming*, 38(2):243–257, 1999.