but not snails (8–11). Normally, caterpillars and snails like to eat some plants (12–13). Every animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants (14). *Conclusion*: there is an animal that likes to eat a grain-eating animal.

The ELP in Fig. 2 has one GL($\leftarrow_L$)-answer set S. This answer set contains $l(b_1, g_1)$, $l(w_1, b_1)$ and $l(f_1, b_1)$, which constitute two constructive solutions to the puzzle (in the sense that $\langle w_1, b_1, g_1 \rangle$ and $\langle f_1, b_1, g_1 \rangle$ are witnesses to the answer). Contrapositive reasoning is essential to the solution, because (14) has to be used in three different directions, for each solution. To illustrate the non-monotonic character of the above formalization, Wagner introduces an abnormal bird O that dislikes caterpillars, and a fox F that likes to eat all smaller animals which are apparently vegetarian.

15. $b(O)$
16. $\neg l(O, x) \leftarrow c(x)$
17. $f(F)$
18. $l(F, x) \leftarrow a(x), sm(x, F), \mathrm{not}\ \neg veg(x)$
19. $\neg veg(x) \leftarrow a(y), l(x, y)$.

Since O dislikes caterpillars, two facts can be deduced. First, there is no evidence that O likes to eat any animal. By (18), it follows that F would like to eat O. Secondly, through (14), it can be concluded that O likes to eat some grain. Thus we get one more solution to the puzzle. Wagner proposes this inference as a benchmark test for automated reasoning systems. The extended ELP consisting of (1)–(19) solves the puzzle. Its unique GL($\leftarrow_L$)-answer set S' contains $l(F, O)$ and $l(O, g_1)$. Note that well-founded Łukasiewicz's semantics yields the same results. More precisely, the least alternating GL($\leftarrow_L$)-answer sets of the two programs are (S, S) and (S', S'), respectively. This is interesting, because this well-founded semantics can be computed efficiently (it can be shown that, for all ground programs P, the least alternating answer set can be computed in time $O(|P|^2)$). Unlike most (all?) of the non-monotonic formalisms in PTIME, the well-founded semantics based on Łukasiewicz's logic is able to solve complex reasoning problems involving incomplete information.

A common objection is that through strong or pseudo negation (and hence answer sets, WFSX, etc.) one can easily solve the same benchmark problems, by explicitly adding the contrapositives of program rules; moreover, this solution is claimed to be more flexible, since programmers can decide whether they want contrapositives or not. This objection disregards the computational cost of the proposed solution, which may cause a quadratic blow-up of space and time complexity. Therefore, this solution is not acceptable in practice for large programs. Secondly, adopting Łukasiewicz's semantics causes no loss of flexibility. When a one-way rule "if B then A" is really needed (although I know of no really convincing motivation for such rules) then it suffices to rewrite the rule as $A \leftarrow B, B$. We conclude that, in general, removing contrapositives when they are not needed is better than adding contrapositives when they are needed.

3.2 Reflexive Transformation

The Gelfond-Lifschitz transformation removes all the program rules which cannot produce any interesting consequence when implication is interpreted as $\leftarrow$. However, when the truth tables by Kleene and Łukasiewicz are adopted, these rules may produce interesting conclusions. It is interesting to study a different transformation which turns each ELP into a monotonic ELP without removing any rule.

Definition 3.8 The *reflexive transformation* of P with respect to S, denoted by P^S_{RE}, is obtained from P by deleting all literals $\mathrm{not}\, L$ such that $L \notin S$, and by replacing each literal $\mathrm{not}\, L$ such that $L \in S$ with $\overline{L}$.

The name "reflexive" is due to a deep analogy with reflexive expansions. If not is interpreted as $\neg L$, then (1) can be rephrased as

$$E = \{\, \varphi \mid \mathrm{AE}(P) \cup \{\, \mathrm{not}\, \psi \mid \psi \notin E \,\} \cup \{\, \mathrm{not}\, \psi \leftrightarrow \neg\psi \mid \psi \in E \,\} \vdash \varphi \,\}.$$

Clearly, P^E_{RE} is nothing but a simplification of AE(P) using $\{\, \mathrm{not}\, \psi \mid \psi \notin E \,\}$ and $\{\, \mathrm{not}\, \psi \leftrightarrow \neg\psi \mid \psi \in E \,\}$. Not surprisingly, we obtain the following result.

Theorem 3.9 *For all ELP's P, S is an RE($\leftarrow_K$)-answer set of P iff* AE(P) *has a reflexive expansion T such that* LIT(T) $= S$.

Alternating RE($\leftarrow_K$)-answer sets do not correspond to any semantics for extended logic programs proposed in the literature.

Example 3.10 Consider the normal program $P = \{\, A \leftarrow_K \mathrm{not}\, B,\ B \leftarrow_K \mathrm{not}\, A,\ C \leftarrow_K A,\ C \leftarrow_K \mathrm{not}\, A \,\}$. While the well-founded model of P corresponds to the alternating sets $(\emptyset, \{\, A, B, C \,\})$, the least alternating answer set of P is $(\{C\}, \{\, A, B, C \,\})$. Note that C is derived although neither A nor not A are derivable; as expected, Kleene's semantics supports nonconstructive inferences.

Next we focus on $\Leftarrow$. Of course, since $\Leftarrow$ is similar to an inference rule, we obtain a semantics which is very similar to standard answer sets.

Theorem 3.11 *For all ELP's P, there is a one to one correspondence between consistent RE($\Leftarrow$)-answer sets of P and answer sets of P.*

The difference between RE and GL becomes evident when alternating answer sets are considered.

Example 3.12 Let $P = \{\, A \Leftarrow \mathrm{not}\, B,\ \neg A \Leftarrow \mathrm{not}\, C,\ B \Leftarrow \mathrm{not}\, C,\ \neg C \,\}$. The least alternating GL($\Leftarrow$)-answer set is $(\{\, \neg C \,\}, \mathrm{LIT})$, while the least alternating RE($\Leftarrow$)-answer set is (S, S), where $S = \{\, \neg C, B, \neg A \,\}$. Here, RE translates not C into $\neg C$; so, from $\neg C$ we can derive $\neg A$ and B. The effect is similar to the coherence principle of WFSX (from $\neg C$ infer not C). In fact, the least alternating RE($\Leftarrow$)-answer set

is always equivalent to WFSX, when the second element of the former is not LIT. In general, however, WFSX is stronger. Consider $P' = \{ A \Leftarrow \text{not}\, \neg A, \neg A \Leftarrow \text{not}\, A, B \Leftarrow \text{not}\, C, \neg C, D \Leftarrow \text{not}\, E \}$. The least alternating RE($\Leftarrow$)-answer set of P' is $M = (\{ \neg C, B \}, \text{LIT})$. WFSX allows to conclude also not E and D.

For normal programs, one can prove that the least alternating RE($\Leftarrow$)-answer sets capture exactly the well-founded semantics.

Finally, we consider Łukasiewicz's implication. RE($\leftarrow_L$)-answer sets are in one-to-one correspondence with the generalized stable models of [11], and hence, they provide the dependency-directed backtracking mechanism of justification-based TMSs [6] with a logical characterization.

Theorem 3.13 *Let P be a normal logic program with constraints. There is a one to one correspondence between consistent RE($\leftarrow_L$)-answer sets of P and generalized stable models of P.*

Thus, we can give a logical explanation of an apparently non-logical step in the transformation by Giordano and Martelli, namely, the elimination of the clauses which are not strictly satisfied.

Example 3.14 Let $P = \{ p \leftarrow \text{not}\, p \}$. The GM transformation must remove the unique rule of P, otherwise p would be a classical consequence of $P^{\{p\}}_{\text{GM}}$, and hence $\{p\}$ would be a GSM of P, while there is no TMS labelling for P. The rule is actually removed, since $p \leftarrow \neg p$ is not strictly satisfied by $\{p\}$. However, there is no independent semantic motivation for this operation. In Łukasiewicz's logic, p is not a logical consequence of $p \leftarrow \neg p$ (the least model of this rule is the empty interpretation). Consequently, we need not eliminate this rule.

Concerning alternating answer sets, with Łukasiewicz's implication we get a well-founded version of the generalized stable model semantics, very similar to the *restricted belief revision model* of [18]. The two semantics agree on all the examples mentioned in [18]. However, in general, they do not coincide.

Example 3.15 Let P be the program with constraints

$$A \quad B_1 \quad C_1 \leftarrow_L A, \text{not}\, B_2 \quad C_2 \leftarrow_L \text{not}\, D \quad \bot \leftarrow_L B_1, B_2 \quad \bot \leftarrow_L C_1, C_2 .$$

The least alternating RE($\leftarrow_L$)-answer set of P corresponds to the RE($\leftarrow_L$)-answer set $S = \{ A, B_1, \neg B_2, C_1, \neg C_2, D \}$, while the restricted belief revision model of P corresponds to $M = (\{ A, B_1, \neg B_2 \}, \text{LIT})$.

As noted in [18], the meaning of strong negation is not always preserved in restricted belief revision models; this motivated a weaker notion called *skeptical belief revision model*. RE($\leftarrow_L$)-answer classes constitute an interesting alternative; the meaning of strong negation is preserved by construction; more inferences can be drawn. Nonetheless, the least alternating RE($\leftarrow_L$)-answer set can be computed in quadratic time, while the skeptical belief revision model requires cubic time.

	$\Leftarrow$	$\leftarrow_L$	$\leftarrow_K$
GL	Default Logic* under tr_1; Answer sets	Idem + Contrapositives	Default Logic* under tr_2; AEL and 3AEL
RE	Consistent answer sets	Gen. stable models as in [11]; DDB in TMS	Reflexive AEL

* Correspondence is with all extension classes.

Table 1: Relations with existing formalisms.

4 Discussion and Conclusions

We have investigated nonmonotonic extension of three-valued logic as a semantic of ELP's. By tuning two parameters, that is, the truth table of implication and the program transformation involved in the nonmonotonic construction, we obtained a unifying view of many formalisms, some of which had never been related before. The main results are summarized in Fig. 1.

This framework highlights surprising similarities between previously unrelated formalisms. The dependency directed backtracking of TMS's can be regarded as a variant of reflexive AEL based on Łukasiewicz's logic. The WFSX semantics by Alferes and Pereira can be regarded as a variant of reflexive AEL based on $\Leftarrow$, modulo the extra non-monotonic inferences supported by the paraconsistent behaviour of WFSX, which make WFSX more powerful.

Łukasiewicz's implication induces some very interesting new semantics, which support contraposition. This apparently minor improvement makes it possible to solve many hard benchmark problems involving incomplete information, with a substantial gain in elegance and efficiency.

Contrapositives model in a natural way the dependency-directed backtracking mechanism of TMS's (Theorem 3.13). The alternating RE($\leftarrow_L$)-amswer sets constitute an appealing alternative to the skeptical belief revision models of [18]. The former are at the same time more powerful and more efficient.

References

[1] A. Avron. Natural 3-valued logics – characterization and proof theory. *The Journal of Symbolic Logic*, 56(1):276-294, (1991).

[2] C.R. Baral, V.S. Subrahmanian. Dualities between Alternative Semantics for Logic Programming and Nonmonotonic Reasoning *The Journal of Automated Reasoning*, 10:399–420, 1993.

[3] C.R. Baral, V.S. Subrahmanian. Stable and Extension Class Theory for Logic Programming and Nonmonotonic Reasoning *The Journal of Automated Reasoning*, 8:345–366, 1992.

[4] P.A. Bonatti. On the monotonic fragment of extended logic programs. Submitted.

[5] P.A. Bonatti. Autoepistemic logics as a unifying framework for the semantics of logic programs. *The Journal of Logic Programming*, 22(2):91-149 (1995).

[6] J. Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12:231–272, (1979).

[7] D.M. Gabbay, F. Guenthner. *Handbook of philosophical logic, Vol. III: Alternatives to classical logic.* Reidel Publishing Co., Dordrecht, Holland, 1986.

[8] D.W. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland, Amsterdam, 1978.

[9] M. Gelfond. On stratified autoepistemic theories. In *Proc. AAAI-87*, pages 207–211, 1987.

[10] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proc. ICLP-90*, pages 579–597, Jerusalem, 1990.

[11] L. Giordano and A. Martelli. Generalized stable models, truth maintenance and conflict resolution. In *Proc. ICLP-90*, pages 427–441, Jerusalem, 1990.

[12] W. Lukasiewicz. *Non-Monotonic Reasoning.* Ellis Horwood Limited, Chichester, England, 1990.

[13] W. Marek and M. Truszczyński. *Nonmonotonic Logics – Context-Dependent Reasoning.* Springer Verlag, 1993.

[14] W. Marek and M. Truszczyński. Stable semantics for logic programs and default theories. In *Proc. of NACLP 89*, pages 243–256, 1989. Springer Verlag, 1993.

[15] F.J. Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191-216, (1986).

[16] L.M. Pereira, J.J. Alferes. Well Founded Semantics for Logic Programs with Explicit Negation. In B. Neumann (ed.), *Proc. of ECAI-92*, pages 102–106, John Wiley & Sons, Chichester, 1992.

[17] G. Schwarz Autoepistemic logic of knowledge. In *Proc. Logic Programming and Non-monotonic Reasoning*, pages 260–274, MIT Press, 1991.

[18] C. Witteveen and G. Brewka. Skeptical reason maintenance and belief revision. *Artificial Intelligence*, 61:1–36, 1993.

[19] G. Wagner. Solving the Steamroller and other puzzles with extended disjunctive logic programs. Proc. of the ICLP-94 Workshop on Non-Monotonic Extensions of Logic Programming, S. Margherita Ligure, Italy, 1994.

Constructing Logic Programs with Higher-Order Predicates [1]

Jørgen Fischer Nilsson
Department of Computer Science
Technical University of Denmark

Andreas Hamfelt
Computing Science Department
Uppsala University

Abstract: This paper proposes a logic programming approach based on the application of a system of higher-order predicates put at disposal within ordinary logic programming languages such as PROLOG. These higher-order predicates parallel the higher-order functionals or combinators which form an established part of contemporary functional programming methodology.

The suggested toolbox of higher-order predicates for composing logic programs is derived from one universal higher-order predicate. They take the form of recursion operators (in particular for expressing recursion along lists) intended to cover all commonly occurring recursion schemes in logic programming practice. Their theoretical sufficiency is proved and their practical adequacy is argued through examples.

The recursion operators, denoting higher-order relations rather than functions, are brought about straightforwardly through a well-known metalogic programming technique, rendering superfluous the need for special higher-order unification mechanisms.

Keywords: Relational higher-order and metalogic programming. Logic program recursion schemes. Declarative logic programming methodology.

Perchè nessuna cosa si può amare nè odiare,
se prima nō si à cognitiō di quella.
– Leonardo da Vinci, *Notebooks*

1 Introduction: Background and Objectives

Higher-order functions (functionals) form a well-established part of the functional programming methodology. It is also well-known how to obtain corresponding restricted higher-order facilities in logic programming. Nevertheless the use of higher-order relations (as opposed to functions) is yet in its infancy as a logic programming methodology.

We propose some higher-order universal predicates and outline a logic programming methodology utilizing these predicates as a programming tool and environment at

[1] Information about authors:
Jørgen Fischer Nilsson: Department of Computer Science 344, DTU, DK-2800 Lyngby, Denmark. Fax: + 45 42 88 45 30. Phone: + 45 45 25 37 30. Email: jfn@id.dtu.dk.
Andreas Hamfelt: Computing Science Department, Uppsala University, Box 311, S-751 05 Uppsala, Sweden. Fax: +46 18 52 12 70. Phone: +46 18 18 10 37. Email: hamfelt@csd.uu.se.

disposal directly within ordinary first-order logic programming in PROLOG. Contrast with dedicated logic programming languages such as λPROLOG [5] and others.

Functional programming as a matter of routine applies higher-order functions for expressing recurring recursions, e.g., list recursions. Such functionals can be epitomized by *fold* which comes in two well-known versions: *foldl(eft)* (also called "accumulate") and *foldr(ight)* ("reduce") both applying to lists and taking as arguments a binary function becoming here a tenary relation P and a constant A.

These higher-order functions translate, respectively, into the higher-order predicates

$$foldl(P, A, nil, A).$$
$$foldl(P, A, B.X, W) \leftarrow P(B, A, V) \wedge foldl(P, V, X, W).$$

and

$$foldr(P, A, nil, A).$$
$$foldr(P, A, B.X, W) \leftarrow foldr(P, A, X, V) \wedge P(B, V, W).$$

1.1 Higher-Order Relations and Logic Programming

In [7] David Warren gave a recipe for emulating higher-order predicates (and functions) as "first class objects" in PROLOG: In essence the higher-order atomic formula $P(t_1, \ldots, t_n)$ is handled as the first-order atomic formula

$$apply(P, t_1, \ldots, t_n)$$

where *apply* is an *ad hoc* predicate expressing predication, that is to say, application of a predicate to its term arguments, defined by clauses

$$apply(p, X_1, \ldots, X_n) \leftarrow p(X_1, \ldots, X_n). \text{ etc.}$$

for each applicable predicate p. These clauses are assumed implicit henceforth and for simplicity's sake we often leave out $apply(\ldots)$ or just write $a(\ldots)$.

Using this simple metalogical term-encoding of predicates, for instance the higher-order predicate *foldr* may be defined by the ordinary PROLOG program

$$foldr(P, Z, nil, Z).$$
$$foldr(P, Z, X.U, W) \leftarrow foldr(P, Z, U, V), a(P, X, V, W).$$

1.2 A Versatile Recursion Operator for Lists

Appealing to the quasi-higher-order predicate *foldr* the ubiquitous *append* can be (re)defined as

$$append(U, V, W) \leftarrow foldr(cons, V, U, W).$$

introducing the auxiliary list constructor predicate *cons* by $cons(X, U, X.U)$.

Folding and unfolding of this definition yield

$$append(nil, U, U).$$
$$append(X.U, V, W) \leftarrow append(U, V, Z), cons(X, Z, W).$$

and the usual form of *append* is now obtained by further unfolding of $cons(X, Z, W)$.

With *append* available, the naïve reverse *rev* can be defined as

$$rev(U, V) \leftarrow foldr(intail, nil, U, V).$$
$$intail(X, U, V) \leftarrow append(U, X.nil, V).$$

from which the usual forms can be regained by partial evaluation or (un)folding.

The efficient ("non-naïve") form of *reverse* (with the accumulator) is obtained from *foldl* simply as

$$reverse(U, V) \leftarrow foldl(cons, nil, U, V).$$

cf. the usual (slightly rewritten)

$$reverse(U, V) \leftarrow rev1(U, V, nil).$$

$$rev1(nil, U, U).$$
$$rev1(X.T, U, V) \leftarrow rev1(T, U, W), cons(X, V, W).$$

This illustrates the replacing of recursive predicate definitions with *prima facie* non-recursive ones. We endeavour eventually to devise predicates endorsing non-recursive formulations of all commonly occuring program predicates.

2 A Hierarchy of Higher-Order Predicates

2.1 The Universal Operator

As the most general operator is suggested

$$\begin{array}{l} univ((Base, Pre1, Post1, Pre2, Post2), X) \leftarrow\ Base(X). \\ univ((Base, Pre1, Post1, Pre2, Post2), X) \leftarrow \\ \qquad Pre1(X, X1), \\ \qquad univ((Base, Pre1, Post1, Pre2, Post2), X1), \\ \qquad Post1(X1, X). \\ univ((Base, Pre1, Post1, Pre2, Post2), X) \leftarrow \\ \qquad Pre2(X, X1, X2), \\ \qquad univ((Base, Pre1, Post1, Pre2, Post2), X1), \\ \qquad univ((Base, Pre1, Post1, Pre2, Post2), X2), \\ \qquad Post2(X1, X2, X). \end{array}$$

From this universal operator in the top of our generalization hierarchy three more practical recursion combinators derive as specializations.

2.2 Fold Operators for Lists

The *foldr* already introduced is obtained as a specialization of *univ*

$$\boxed{\begin{array}{l} foldr(P,Z,nil,Z). \\ foldr(P,Z,X.U,W) \leftarrow foldr(P,Z,U,V), a(P,X,V,W). \end{array}}$$

$a(foldr,P,Z,U,V) \leftarrow a(univ(base(Z),pre,post(P),fail,fail),(U,V)).$
$a(base(Z),(nil,Z)).$
$a(pre,(X.U,W),(U,V)).$
$a(post(P),(\_,V),(X.\_,W)) \leftarrow a(P,X,V,W).$

2.3 Linear Recursion Operator

From *foldl* or from *univ* can be obtained another useful operator

$$\boxed{\begin{array}{l} linrec((P,Q),X,Y) \leftarrow P(X,Y). \\ linrec((P,Q),X,Z) \leftarrow Q(X,Y), linrec((P,Q),Y,Z). \end{array}}$$

$a(linrec(P,Q),X,Y) \leftarrow a(univ(base(P),pre1(Q),true,fail,fail),X,Y).$
$a(base(P),(X,Y) \leftarrow a(P,X,Y).$
$a(pre1(Q),(X,Z),(Y,Z)) \leftarrow a(Q,X,Y).$

This operator may be used, e.g., for defining the transitive closure of a binary relation r simply through: $a(closure(r),X,Y) \leftarrow a(linrec(r,r),X,Y)$.

As another example the commonplace membership can be expressed simply as

$$member(X,L) \leftarrow linrec((head,tail),L,X).$$
$$\left\{\begin{array}{l} head(X.T,X). \\ tail(X.T,T). \end{array}\right.$$

Notice that the operator provides unbounded recursion in contrast to *fold*, which is bound to terminate when applied to ground lists. As discussed below this recursion operator suffices theoretically in the sense that it covers all computable functions. However, pragmatical adequacy calls at least also for the following operator.

2.4 Divide-and-conquer Operator

$$\boxed{\begin{array}{l} divconc((Base,Decomp,Comp),X,Y) \leftarrow Base(X,Y). \\ divconc((Base,Decomp,Comp),X,Z) \leftarrow \\ \qquad Decomp(X,X1,X2), \\ \qquad divconc((Base,Decomp,Comp),X1,Y1), \\ \qquad divconc((Base,Decomp,Comp),X2,Y2), \\ \qquad Comp(X,Y1,Y2,Z). \end{array}}$$

The quicksort algorithm in the simple ordinary PROLOG formulation (without difference lists), cf. e.g. [6], lends itself to formulation by means of the divide-and-conquer operator:

$$qsort(L,Ls) \leftarrow divconc((emptycase,decomp,comp),L,Ls).$$
$$\left\{\begin{array}{l} emptycase(nil,nil). \\ decomp(X.T,L1,L2) \leftarrow partition(leq(X),gr(X),L,L1,L2). \\ \qquad\qquad \left\{\begin{array}{l} a(leq(X),Y) \leftarrow \quad Y \leq X. \\ a(gr(X),Y) \leftarrow \quad Y > X. \end{array}\right. \\ comp(X.T,L1,L2,L3) \leftarrow append(L1,X.L2,L3). \end{array}\right.$$

The higher-order predicate *partition* (like *append*) is definable straightforwardly using *foldr*. The brackets serve to indicate the purely hierarchical structure of the program having its recursions embraced in the operators.

3 Adequacy of Proposed Recursion Operators

In order to bring to bear the higher-order operator technique as a general logic programming methodology we have to argue that the above recursion operators suffice e.g. for the collection of (pure) logic programs in [6]. Most of the programs therein can be managed using solely *fold* and *linrec*, more intricate pure programs calls for *divconq* or even the most general *univ* as in the case of meta-interpreters. This investigation will be reported in a forthcoming paper in preparation.

In order to prove that the *linrec* recursion is universal it suffices to devise a definite clausal logic program emulating the universal Turing machine. This establishes the theoretical universality of the *linrec* scheme and a *fortiori* the *univ* operator.

4 Program Refinement

As an example of a program refinement technique let us consider how to cope with difference lists intended to improve program efficiency by a low cost concatenation. This is common logic programming practice for instance on the output list of *quicksort*, cf. e.g. [6].

We want to devise a generalised and principled approach which avoids elaborate program alterations. Considering again the *divconc* recursion scheme to this end we suggest refinement with difference lists solely through the parameter predicates to the higher-order predicate, witness:

$$a(divconcdf(B,D,C),X,Y) \leftarrow dfversion(B,Bdf),dfversion(C,Cdf),$$
$$\qquad\qquad a(divconc(Bdf,D,Cdf),X,(Y,nil)).$$

where *dfversion* is a library catalog providing corresponding difference list versions

of various list processing predicates.

In this way higher-order parameterized predicate formulations facilitate program refinements avoiding comprehensive and often rather incomprehensible transformations throughout the programs.

5 Summary

We have introduced a generalization hierarchy of recursion operators in the form of higher-order predicates:

$$univ \rightarrow divconc \rightarrow foldl,\ foldr \rightarrow linrec$$

In a forthcoming paper in preparation we consolidate the theoretical adequacy by relating to recursive function theory, whereas the practical potentialities of the chosen set of operators – especially with respect to obtaining declarative readings – are studied on a corpus of common logic programs.

References

[1] T.S. Gegg-Harrison: Basic Prolog Schemata, CS-1989-20, Dept. of Computer Science, Duke University, 1989.

[2] A. Hamfelt & J. Fischer Nilsson: Inductive Metalogic Programming, *Procs. of Workshop on Inductive Logic Programing*, S. Wrobel (ed.), Bad/Honnef/Bonn 1994, GMD-Studien Nr. 237, ISSN 0170-8120, 1994.

[3] M. Kirschenbaum & L. S. Sterling: Refinement Strategies for Inductive Learning of Simple Prolog Programs, *Procs. of the 12th Int. Conf. on Artificial Intelligence, IJCAI-91*, Sydney, 1991. pp. 757-761.

[4] E. Marakakis & J. P. Gallagher: Schema-Based Top-Down Design of Logic Programs Using Abstract Data Types, *Logic program Synthesis and Transformation – Meta-Programming in Logic*, LOPSTR'94 and Meta'94, L. Fribourg & F. Turini (eds.), LNCS 883, Springer, 1994.

[5] D.A. Miller & G. Nadathur: Higher-order Logic Programming, *Procs. of the 3rd International Logic Programming Conference*, LNCS 225, Springer-Verlag, 1986.

[6] L. Sterling & E. Shapiro: *The Art of Prolog*, MIT Press, 1986, 1991.

[7] D. H. D. Warren: Higher-order extensions to PROLOG: are they needed ?, D. Michie (ed.): *Machine Intelligence 10*, Ellis Horwood and Edinburgh University Press, 1982. pp. 441-454.

Petri Nets and Linear Logic: a case study for logic programming

Iliano Cervesato*

Department of Computer Science – Carnegie Mellon University
5000 Forbes Avenue – Pittsburgh, PA, 15213-3891
Phone: (412)-268-1413 – Fax: (412)-268-5576
iliano@cs.cmu.edu

Abstract

The paper reports on an experiment with the major linear logic programming languages defined in the recent years: Lolli, LO and Forum. As a case study, we consider the representation of a class of Petri nets, P/T^{ω} nets, in each of these languages.

Keywords: Petri nets, multiset rewriting systems, linear logic, logic programming.

1 Introduction

The relationship between linear logic [4] and Petri Nets [8] has been a major object of investigation [2, 3, 6]. It was soon noticed that the monoidal structure of multisets can serve as an abstract link between Petri nets (where markings are multisets and transition are rewrite rules on multisets) and the multiplicative fragment of linear logic (where either $\otimes$ or $\parr$ is the operation of the monoid and $\mathbf{1}$ or $\bot$ is its identity). See [3] for a precise account of this relationship.

On the other hand, a number of fragments of linear logic have been cast into logic programming languages with the aim of capturing some of the expressive power provided by this new logic. We consider three such proposals, Lolli [5], LO [1] and Forum [7], and compare their flexibility precisely on the encoding of Petri nets (or equivalently of multiset rewriting systems). Considerations about the amenability of these languages to efficient implementations are outside the scope of this paper.

*This research was completed during an extended visit of the author at the *Department of Computer Science* of *Carnegie Mellon University*. His permanent address is: *Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy.*

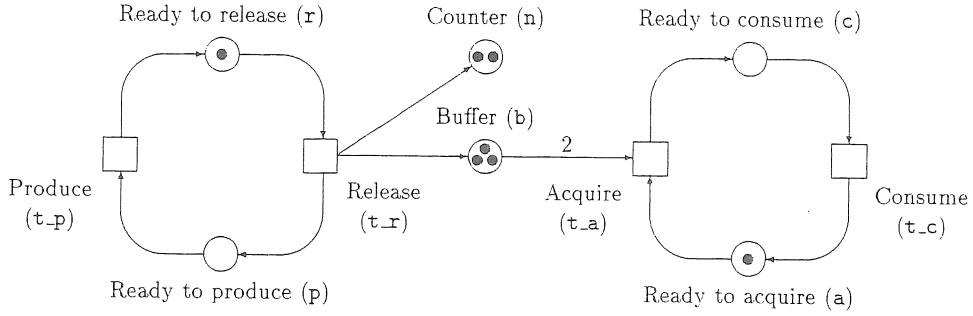

Figure 1: A producer/consumer net

2 P/T$^\omega$ nets

A *multiset rewriting system* (*MRS*) $\mathcal{M}$ over a set S is a set of *multiset rewrite rules* $t = {}^\bullet t \triangleright t^\bullet$ with ${}^\bullet t$ and $t^\bullet$ multisets over S called respectively the *preset* and the *postset* of t. Given a multiset M, a rule t in $\mathcal{M}$ is *enabled* at M if ${}^\bullet t$ is a submultiset of M. If t is enabled at M, the *application* of t on M produces the multiset $M' = M - {}^\bullet t + t^\bullet$, $+$ and $-$ being the multiset union and difference respectively. We write in this case $M \triangleright^t_{\mathcal{M}} M'$ (or $M \triangleright_{\mathcal{M}} M'$). We use $\triangleright^*_{\mathcal{M}}$ for the reflexive and transitive closure of $\triangleright_{\mathcal{M}}$.

A *place/transition net with infinite place capacities*, *P/T$^\omega$ net* for short, is a pair $N = (S, \mathcal{M})$ where S is a set and $\mathcal{M}$ is an MRS over S. We require either S or $\mathcal{M}$ to be non-empty. The elements of S are called *places* and the elements of $\mathcal{M}$ are called *transitions*. A *P/T$^\omega$ system* is a pair $\mathcal{N} = (N, M)$ where $N = (S, \mathcal{M})$ is a P/T$^\omega$ net and M is a multiset over S called a *marking*.

Figure 1 describes a producer-consumer system by means of the usual graphical representation for Petri nets. The left cycle represents the producer, that releases one item per cycle, puts it in the common buffer and increments by one the counter. The consumer (on the right) extracts two items at a time from the buffer and consumes them. We will use this example as a benchmark for our implementations. In section 4, we will rely on the abbreviations shown in parentheses.

The dynamic behavior of a P/T$^\omega$ net is inherited from the underlying MRS. In this context, the application of an enabled transition (i.e. a multiset rewrite rule) to a marking is referred to as the *firing* of that transition.

3 Encoding Petri nets in linear logic

We describe in this section two encodings of P/T$^\omega$ systems in multiplicative linear logic. The first uses the language $\mathcal{L}^\otimes$ to encode multisets by means of the connectives $\otimes$ and $\mathbf{1}$. Dually, $\mathcal{L}^{\parr}$ relies on $\parr$ and $\bot$.

A sequent for $\mathcal{L}^\otimes$ has the form $\Gamma; \Delta \vdash C$ where C is a possibly empty conjunction of atomic formulas (C-formula), Δ is a multiset of C-formulas and Γ is a set of linear implications with C-formulas as antecendent and consequent (I-formula). The rules

$$\frac{}{\Gamma; C \vdash C}\ \text{axiom}' \qquad \frac{\Gamma; \Delta \vdash C}{\Gamma; \Delta, \mathbf{1} \vdash C}\ \mathbf{1}\text{L}' \qquad \frac{}{\Gamma; \cdot \vdash \mathbf{1}}\ \mathbf{1}\text{R}'$$

$$\frac{\Gamma; \Delta, C_1, C_2 \vdash C}{\Gamma; \Delta, C_1 \otimes C_2 \vdash C}\ \otimes\text{L}' \qquad \frac{\Gamma; \Delta_1 \vdash C_1 \quad \Gamma; \Delta_2 \vdash C_2}{\Gamma; \Delta_1, \Delta_2 \vdash C_1 \otimes C_2}\ \otimes\text{R}'$$

$$\frac{\Gamma, C_1 \multimap C_2; \Delta_1 \vdash C_1 \quad \Gamma, C_1 \multimap C_2; \Delta_2, C_2 \vdash C}{\Gamma, C_1 \multimap C_2; \Delta_1, \Delta_2 \vdash C}\ \multimap\text{L}'$$

Figure 2: The system MILL$^{P/T}$ for $\mathcal{L}^\otimes$.

for the resulting system are presented in figure 3. MILL$^{P/T}$ is proved in [3] to be equivalent to the usual system for full linear logic when restricted to formulas in $\mathcal{L}^\otimes$.

In order to represent P/T$^\omega$ nets in $\mathcal{L}^\otimes$, we associate to every element in the universe of a multiset a propositional letter. A multiset is then mapped to a C-formula consisting of the multiplicative conjunction of its members. A rewrite rule is represented by an I-formula built upon the representation of its preset and postset. We will call this representation the *conjunctive encoding* and denote it with $\ulcorner\cdot\urcorner$, being $\llcorner\cdot\lrcorner$ the inverse function. We have the following result [3].

Theorem 1 (*Soundness and completeness of the conjunctive encoding of MRSs*)
Let M and M' be two multisets and $\mathcal{M}$ a multiset rewriting system over a common universe. Then, if $M \triangleright^*_{\mathcal{M}} M'$, there exist a derivation of $\ulcorner\mathcal{M}\urcorner; \ulcorner M\urcorner \vdash \ulcorner M'\urcorner$.

Let Γ, Δ and C be an I-context, a C-context and a C-formula respectively, Then, if $\mathcal{PT}$ is a derivation of $\Gamma; \Delta \vdash C$, then $\llcorner\Delta\lrcorner \triangleright^*_{\llcorner\Gamma\lrcorner} \llcorner C\lrcorner$. ∎

The dual representation in $\mathcal{L}^{\parr}$ consists in mapping multisets to D-formulas, i.e. the set of formulas freely generated from $\bot$ and $\parr$. We call this representation the *disjunctive encoding* P/T$^\omega$ systems [3].

4 Implementations in linear logic programming

We will now consider in turn three logic programming languages based on linear logic and show how the notions presented in the previous section can be effectively implemented by using each of them. The languages we chose are Lolli [5], LO [1] and Forum [7]. These three languages differ for the fragment of linear logic they are based upon. Consequently, the encoding of MRSs, and therefore of P/T$^\omega$ systems, will be different in each language. We foresay that the implementation will become more direct and elegant as we move from Lolli to LO and finally to Forum. As a benchmark, we use the producer-consumer example of figure 1.

4.1 Implementation in Lolli

The fragment of linear logic that Lolli [5] makes available as a programming language is the first-order language freely generated from $\top$, $\&$, $\multimap$, $\Rightarrow$, where $A \Rightarrow B$ is

Generic rules

```
rewrite L K :- {load L K}.
load (X::L) K :- (item X -o load L K).
load nil    K :- rew K.
rew K :- unload K.
unload (X::L) :- item X, unload L.
unload nil.
```

Specific rules

```
rew K :- item p, (item r -o rew K).                                  % t_p
rew K :- item r, (item p -o (item b -o (item n -o rew K))).          % t_r
rew K :- item b, item b, item a, (item c -o rew K).                  % t_a
rew K :- item c, (item a -o rew K).                                  % t_c
```

Example query

```
?- rewrite r::n::n::b::b::b::a::nil  p::n::n::n::b::c::nil.
```

Figure 3: A Lolli encoding of the producer/consumer net

defined to be equal to $!A \multimap B$, and $\forall$[1]. Lolli formalizes intuitionistic provability for this fragment of linear logic over sequents of the form $\Gamma; \Delta \vdash G$, where Γ and Δ are two multisets of formulas called the *unbound* and the *bound context* respectively and G is the *goal formula*.

For our purpose, we will rely exclusively on the connectives $\otimes$, $\multimap$ and ! (written respectively `,`, `-o` or `:-`, and $\{\ldots\}$ in the concrete syntax). Operationally, $\otimes$ requires the bound context to be split among its two subgoals, $D \multimap G$ involves adding D to the bound context in order to prove G, and $!G$ succeeds if G is provable in an empty bound context. When a clause or a fact from the bound context is used, it is deleted.

Lolli does not provide the tools for a direct implementation of neither the conjunctive nor the disjunctive encoding. Therefore, an elegant solution cannot be achieved within Lolli. However, we can simulate the former by representing a generic rule $t = \{\!\{a_1, \ldots, a_n\}\!\} \triangleright \{\!\{b_1, \ldots, b_m\}\!\}$ as the clause `rew K :- a1, ..., an, (b1 -o (b2 -o ... (bm -o rew K)...))`. When called, the first part of this clause (`a1, ..., an`) consumes the preset of t while the remaining part asserts its postset and finally calls `rew K` recursively for the application of another rule.

The full program, adapted from [5], is shown in figure 3. Markings are given two representations: the users views them externally as lists and the program represents internally their constituents as individual facts in the bound context. A multiset rewriting sequence is performed in three stages: first the list representing the initial marking is downloaded into the bound context (`load`), then the rewrite rules are applied (`rew`), and finally the resulting multiset is encoded back into a list (`unload`).

[1]The syntax of Lolli can be enriched by permitting the following connectives in a goal position: $\oplus$, **1**, $\otimes$, ! and $\exists$. The resulting formulas are called (linear) *hereditary Harrop formulas*.

Specific rules

```
r          :- p.                          % t_p
p @ b @ c  :- r.                          % t_r
c          :- b @ b @ a.                  % t_a
a          :- c.                          % t_c
r @ n @ n @ b @ b @ b @ a        @ bot @ stop.
```

Example query

```
?- p @ n @ n @ n @ b @ c         @ bot @ stop.
```

Figure 4: An LO encoding of the producer/consumer net

4.2 Implementation in LO

LO mechanizes the fragment of linear logic specified by the following grammar rules:

$$V ::= A \mid V_1 \parr V_2 \qquad \text{(Head formulas)}$$
$$G ::= A \mid \top \mid G_1 \,\&\, G_2 \mid G_1 \parr G_2 \qquad \text{(Goal formulas)}$$
$$D ::= V \mid G \multimap V \qquad \text{(Clauses)}$$

where A ranges over atomic formulas (for the purpose of our example, it will be sufficient to deal with the propositional fragment of LO).

The operational semantics of LO is conveniently described by means of classical sequents of the form $\Gamma \vdash \Theta$, where Γ is a set of clauses called the *program* and Θ is a multiset of goal formulas called the *context*. The dynamic part of an LO sequent is its context, that will in general contain more than one goal. Once all the goals have been decomposed, a clause is fetched from the program and used to continue the computation. In the following, we will deal with goal formulas containing multiplicative disjunctions (written `@` in LO) of atoms only.

Having multiplicative disjunction built-in, LO is adequate for coding the disjunctive representation of P/T$^\omega$ nets. We represent places as atomic formulas, use multiplicative disjunction to encode markings and a clause of the form `b1 @ ... @ bm :- a1 @ ... @ an` for each transitions $\{\!\{a_1, \ldots, a_n\}\!\} \triangleright \{\!\{b_1, \ldots, b_m\}\!\}$. Since $\bot$ is not available in LO, we simulate it by means of the atomic formula `bot`, making it play the role of the empty multiset whenever a rule has an empty postset. Since LO does not possess a bound context, we encode the initial marking by means of a rule with no postset, distinguished by the atom `stop`.

Figure 4 shows the resulting program for the producer/consumer example.

4.3 Implementation in Forum

Forum [7] was designed as an extension to both Lolli and LO. The fragment of linear logic it mechanizes is the language freely generated from the linear logic symbols $\top$, $\&$, $\bot$, $\parr$, $\multimap$, $\Rightarrow$ and $\forall$. This set of connectives is indeed complete in the sense that the remaining linear symbols can be defined on top of it. The operational semantics of Forum is modelled by means of sequents of the form $\Gamma; \Delta \vdash \Theta$ where the functions of the contexts Γ, Δ and Θ coincide with those of the homonynous entities of Lolli and LO.

```
                                                        Specific rules
r          :- p.                        % t_p
p @ b @ c  :- r.                        % t_r
c          :- b @ b @ a.                % t_a
a          :- c.                        % t_c
LINEAR r @ n @ n @ b @ b @ b @ a.
                                                        Example query
?- p @ n @ n @ n @ b @ c.
```

Figure 5: A Forum encoding of the producer/consumer net

Figure 5 shows the Forum implementation of the producer/consumer example. In this case, we can adopt in full the disjunctive encoding. The resulting program resembles the LO implementation. Since Forum admits $\bot$ as a primitive connective, there is no need for any trick to simulate the behavior of the empty multiset. Moreover, since there is a distinction between a bound and an unbound part of the program, a termination predicate like `stop` is not required in this implementation. The initial marking is loaded in the bound context by means of the directive `LINEAR`.

5 Conclusions

In this paper, we reported on an experiment with the major linear logic programming languages designed in the recent years: Lolli [5], LO [1] and Forum [7]. As a case study, we considered the representation of a class of Petri nets, P/T^{ω} nets, in each of these languages. We observed that the obtained code becomes more direct and elegant as we pass from Lolli to LO, and from LO to Forum. This fact is easily explained by noticing that the set of linear connectives provided by the languages in this list becomes more and more suited to our problem as we proceed.

References

[1] J.M. Andreoli, R. Pareschi: "Linear Objects: Logical Processes with Built-in Inheritance", in *International Conference on Logic Programming* pp. 495–510, 1990.

[2] A. Asperti: "A logic for Concurrency", Manuscript, November 1987.

[3] I. Cervesato: "Petri Nets as Multiset Rewriting Systems in a Linear Framework". Available as http://www.cs.cmu.edu/~iliano/ON-GOING/deMich.ps.gz.

[4] J.Y. Girard: "Linear Logic", in *Theoretical Computer Science* 50:1–102, 1987.

[5] J. Hodas, D. Miller: "Logic Programming in a Fragment of Intuitionistic Linear Logic", in *Journal of Information and Computation* 110(2):327–365, 1994.

[6] N. Martí-Oliet, J. Meseguer: "From Petri Nets to Linear Logic", in *Conf. on Category Theory and Computer Science*, pp. 313–337, LNCS 389, Springer-Verlag, 1989.

[7] D. Miller: "A Multiple-Conclusion Meta-Logic", in *Symposium on Logic in Computer Science*, pp. 272–281, 1994.

[8] W. Reisig: "*Petri Nets, an Introduction*", Springer-Verlag, 1985.

GRAMPAL: A Morphological Processor for Spanish implemented in Prolog*

Antonio Moreno
Dep. de Lingüística
Universidad Autónoma de Madrid
Cantoblanco, Madrid, SPAIN
Phone: +34 1 397.41.09
Fax: +34 1 397.39.30
sandoval@ccuam3.sdi.uam.es

José M. Goñi
Dep. Matemática Aplicada a las TT.II.
Universidad Politécnica de Madrid
Ciudad Universitaria, Madrid, SPAIN
Phone: +34 1 336.72.87
Fax: +34 1 336.72.89
jmg@mat.upm.es

Abstract

A model for the full treatment of Spanish inflection for verbs, nouns and adjectives is presented. This model is based on feature unification and it relies upon a lexicon of allomorphs both for stems and morphemes. Word forms are built by the concatenation of allomorphs by means of special contextual features. We make use of standard Definite Clause Grammars (DCG) included in most Prolog implementations, instead of the typical finite-state approach. This allows us to take advantage of the declarativity and bidirectionality of Logic Programming for NLP.

The most salient feature of this approach is simplicity: A really straightforward rule and lexical components. We have developed a very simple model for complex phenomena.

Declarativity, bidirectionality, consistency and completeness of the model are discussed: all and only correct word forms are analysed or generated, even alternative ones and gaps in paradigms are preserved. A Prolog implementation has been developed for both analysis and generation of Spanish word forms. It consists of only six DCG rules, because our *lexicalist* approach –i.e. most information is in the dictionary. Although it is quite efficient, the current implementation could be improved for analysis by using the non logical features of Prolog, especially in word segmentation and dictionary access.

Keywords: Applications of Logic Programming to NLP, Computational Morphology.

*This work has been partially supported by the Spanish *Plan Nacional de I+D*, under the Research Project *An Architecture for para Natural Language Interfaces with User Modeling* (TIC91-0217C02-01).

1 Introduction

The successful treatment of morphological phenomena in some languages by means of finite state automata appears to have led to the idea that this model is the most efficient and universal way to deal with morphology computationally. Although there exist good finite-state processors for Spanish –like [Martí, 1986], [Meya, 1986] or [Tzoukermann & Liberman, 1990]– we think that some phenomena can be handled more elegantly using a context-free approach, particularly if the morphological component is to be included as a part of a syntax grammar. Our model has been implemented in standard DCG using a logic programming approach instead of a plain finite-state one.

It is well-known that the so-called non-concatenative processes are the most difficult single problem that morphological processors must deal with. Experience has shown that it is not easy for any approach. Unification-based morphology uses suppletion (i.e. alternative allomorphs for a lemma) and feature description as a general mechanism for handling those processes. Two-level morphology uses instead rules that match lexical representations (lemmas) with surface representations (actual spelling forms). The latter has been been claimed to be more elegant, but it is obvious that often the two-level model contains many rules needed for a very few cases.

The pure two-level/finite-state automata model is not very adequate for treating certain non-concatenative processes, and in such cases one is required to depart from this approach, for example by adding an extension in which two-level rules are retained under the control of feature structures [Trost, 1990]. Moreover, every language has irregularities that can only be treated as suppletive forms, e.g. `soy` (*I am*), `era` or `fui` (*I was*). Since suppletion is needed anyway, and since it is a much simpler approach than rules, we consider that the "elegance" objection is not well-founded[1].

On the other hand, our goal is to generate and recognize all (and only) well-formed inflected forms, and thus we do not accept "*missing forms*" for defective verbs (see below), but do accept duplicate but correct forms.

2 Major issues in Spanish computational morphology

Spanish morphology is not a trivial subject. As an inflectional language, Spanish shows a great variety of morphological processes, particularly non-concatenative ones. We will try to summarize the outstanding problems which any morphological processor of Spanish has to deal with:

[1] See the next section for further discussion of the adequacy of the two-level model for Spanish, including defective forms (i.e. null forms in the conjugation) and alternative correct forms.

1. A highly complex verb paradigm. For simple tenses, we consider 53 inflected forms (see Table 1), excluding the archaic Future Subjunctive, but including the duplicate Imperfect Past Subjunctive (6 forms). If we add the 45 possible forms for compound tenses, then 98 inflected forms are possible for each verb.

2. The frequent irregularity of both verb stems and endings. Very common verbs, such as `tener` (*to have*), `poner` (*to put*), `poder` (*to be able to*), `hacer` (*to do*), etc., have up to 7 different stems: `hac-er`, `hag-o`, `hic-e`, `ha-ré`, `hiz-o`, `haz`, `hech-o`. This example shows internal vowel modification triggered by different morphemes having the same external form: `hag-o`; `hiz-o`, `hech-o` (The first `/-o/` is first person singular present indicative morpheme; the second `/-o/` is third singular preterit indicative morpheme; and the third `/-o/` is past participle morpheme – an irregular one, by the way). As well as these non-concatenative processes, there exist other, very common, kinds of internal variation, as illustrated by the following example.

   $[e] \rightarrow [ie]$: `quer-er` (*to want*) → `quier-o` (*I want*)

   2,300 out of 7,600 verbs in our dictionary are classified as irregular, and 5,300 as regular –i.e. only one stem for all the forms as in `am-ar` — `am-o`, etc. (*to love*).

3. Gaps in some verb paradigms. In the so-called *defective verbs* some forms are missing or simply not used. For instance, meteorological verbs such as `llover`, `nevar` (*to rain, to snow*), etc. are conjugated only in third person singular. Other ones are more peculiar, like `abolir` (*to abolish*) that lacks first, second and third singular and third plural present indicative forms, all present subjunctive forms, and the second singular imperative form. In other verbs, the compound tenses are excluded from the paradigm, like in `soler` (*to do usually*).

4. Duplicate past participles: a number of verbs have two alternative forms, both correct, like `impreso`, `imprimido` (*printed*). In such cases, the analysis has to treat both.

5. There exist some highly irregular verbs that can be handled only by including many of their forms directly in the lexicon (like `ir` (*to go*), `ser` (*to be*), etc).

6. Nominal inflection can be of two major types: with grammatical gender (i.e. concatenating the gender morpheme to the stem) and with inherent gender (i.e. without gender morphemes). Most pronouns and quantifiers belong to the first class, but nouns and adjectives can be in any of the two classes, with a different distribution: 4% of the nouns have grammatical gender and 92% have inherent gender, while 70% of the adjectives are in the first group. Some nouns and adjectives present alternative correct forms for plural –e.g. for `bambú` (*bamboo*), `bambú-s` and, `bambú-es`.

7. There is a small group (3%) of invariant nouns with the same form for singular and plural, e.g. `crisis`. On the other hand, 30% of the adjectives present the same form for masculine and feminine, e.g. `azul` (*blue*). There exist also *singularia tantum*, where only the singular form is used, like `estrés` (*stress*); and *pluralia tantum*, where only the plural form is allowed, e.g. `matemáticas`, (*mathematics*).

8. In contrast with verb morphology, nominal processes do not produce internal change in the stem caused by the addition of a gender or plural suffix, although there can be many allomorphs produced by spelling changes: `luz`, `luc-es` (*light, lights*).

For a detailed description of all verb and nominal phenomena, including a classification into paradigmatic models, see [Moreno, 1991].

All these phenomena suggest that there is no such a universal model (e.g. two-level, unification, or others) for (surface) morphology. Instead, we have approaches more suited for some processes than others. The computational morphologist must decide which is more appropriate for a particular language. We support the idea that unification and feature-based morphology is more adequate for languages, such as Spanish and other Latin languages, that have alternative stems triggered by specific suffixes, missing forms in the paradigm, and duplicate correct forms.

3 The model

It is well known that morphological processes are divided into two types: processes related to the phonological and/or graphic form (morpho-graphemics), and processes related to the combination of morphemes (morpho-syntax). Each model treats these facts from its particular perspective. Two-level morphology uses phonological rules and continuation classes (in the lexical component). Mixed systems such as [Bear, 1986] or [Ritchie et. al., 1987] have different sets of rules.

As we stated before, our model relies on a context-free feature-based grammar, that is particularly well suited for the morpho-syntactic processes. For morpho-graphemics, our model depends on the storage –or computation– of all the possible allomorphs both for stems and endings. This feature permits that both analysis and synthesis be limited to morpheme concatenation, as the general and unique mechanism. This simplifies dramatically the rule component.

We present some examples of dictionary entries: two verbal ending entries (allomorphs) for the past participle morphemes and two allomorph stems for *imprimir*, compatible with those endings.

```
vm(no,part,nofin,[2,3],99,reg) --> [ido].
vm(no,part,nofin,[2,3],99,part1) --> [o].
```



```
vl(imprimir,v,3,[100],[reg]) --> [imprim].
vl(imprimir,v,3,[99],[part1]) --> [impres].
```

Where `vm` and `vl` stands for the values of the "morphological category" that we are using to drive the DCG rule invocation. All the dictionary entries are coded with a predicate that corresponds to its morphological category. The full inventory of such categories follows:

w For complete inflected word forms.

wl For words (nouns and adjectives) that can accept a number morpheme.

vl For verb lexemes (stems).

nl For nominal –nouns and adjectives– lexemes.

vm For verb morphemes.

ng For nominal gender morphemes.

nn For nominal number morphemes.

For reference, and to check the meaning of the examples, a short self-description of the arguments of those predicates follows:

```
w(Lemma, Category, Pers_Num, Tense_Mood).
w(Lemma, Category, Gender, Number).
wl(Lemma, Category, Number_Type_List, Gender, Number).

vl(Lemma, Category, Conjugation, Stem_Type_List,
   Suffix_Type_List).
vm(Pers_Num, Tense_Mood, Finiteness, Conjugation_List,
   Stem_Type, Suffix_Type).

nl(Lemma, Category, Gender_Type_List, Number_Type_List,
   Gender, Number).
ng(Gender_Type, Gender, Number).
nn(Number_Type, Number).
```

We have introduced some contextual atomic features that impose restrictions on the concatenation of morphemes through standard unification rules. Such features are never percolated up to the parent node of a rule. Multi-valued atomic features are permitted in the unification mechanism, being interpreted as a disjunction of atomic values. We represent this disjunction as a Prolog list. Disjunction of values is used only for contextual features (*stem_type*, *suffix_type*, *conjugation*, *gender_type*

and *number_type*) just to improve storage efficiency, since this device is actually not needed if different entries were encoded in the lexicon.

In the conjugation table (Table 1), the *stem_type* values of the grammatical features person-number and tense-mood are displayed in boldface. For example, **sing_1** means first person, singular number; while **pres_ind** means present tense, indicative mood.

	sing_1	sing_2	sing_3	plu_1	plu_2	plu_3	—
pres_ind	11	12	13	14	15	16	
impf_ind	21	22	23	24	25	26	
indf_ind	31	32	33	34	35	36	
fut_ind	41	42	43	44	45	46	
pres_subj	51	52	53	54	55	56	
impf_subj	61	62	63	64	65	66	
cond	71	72	73	74	75	76	
imper		82	83		85	86	
inf							00
ger							90
part							99

Table 1: Conjugation table.

Each of the 49 inflected forms[2] is represented by a numeric code[3], and the additional value 100 is used as a shorthand for the disjunction of all of them (used for regular verbs; see the entry for `imprim` above). The contextual feature *stem_type* (**stt**) is used to identify the verb stem and ending corresponding to each form, and the contextual feature *suffix_type* (**sut**) distinguishes among several allomorphs of the inflectional morpheme by means of a set of values:

```
reg       pres      pret1  pret2
fut_cond  imp_subj  imper  infin
ger       part1     part2
```

Since this value set is much smaller than the *stem_type set*, we have chosen an alphabetic code. With the combination of both features, and the addition of a third feature `conj` (conjugation), we can state unequivocally which is the correct sequence of stem and ending for each case (see examples above, where `imprim` only matches `ido` for all features, and `impres` matches `o`, thus preventing ill-formed concatenations –for these morphemes– such as `imprim-o` or `impres-ido`).

[2]The codes 83 and 86 stand for the *courtesy* imperative: `imprima usted`, `impriman ustedes`. These word forms are the same as the 53 and 56 ones.

[3]Actually, this number encodes a particular combination of person, number, tense and mood features.

In the same fashion, we have two special contextual features for the nominal inflection, `nut` (number_type) and `get` (gender_type), to identify the various allomorphs for the plural and gender morphemes, and associate them with the proper nominal stems. The following examples show those contextual features both in nominal morphemes and in nominal lexeme entries:

```
/*     NOUN AND ADJECTIVE MORPHEMES     */
ng(mas1,masc,sing) --> [o].
ng(mas2,masc,sing) --> [e].
ng(fem,fem,sing)   --> [a].

nn(plu1,plu) --> [s].
nn(plu2,plu) --> [es].

/*     SOME ENTRIES FOR NOUNS     */
nl(presidente, n, [mas2,fem], [plu1], _, _) --> [president].

wl(doctor, n, [plu2], masc, sing) --> [doctor].
nl(doctor, n, [fem], [no], masc, sing) --> [doctor].

wl(bambu1,  n, [plu1,  plu2],  masc,  sing) --> [bambu1].
```

These entries allow the analysis/generation of the word forms `presidente`, `presidenta`, `presidentes` and `presidentas` for the lemma `presidente`; `doctor`, `doctora`, `doctores` and `doctoras` for `doctor`; and `bambú`, `bambús` and `bambúes` for `bambú`.

The grammatical features (*category, lemma, tense, mood, person, number* and *gender* are the only features that are delivered to the **w** node, and from this level can be used by a syntactic DCG grammar.

A unification-based system relies very much on the lexical side. It is needed a robust and large dictionary, properly coded. Additionally, our model depends on the accessibility of all possible allomorphs, so their storage is also necessary. Fortunately, there is no need for typing all of them by hand, since this would be an impractical, time consuming and error-prone task. Morpho-graphemics for Spanish is quite regular and we have formalized and implemented the automatic computation of the allomorphs of any verb from the infinitive form.

The formalized description of the morphological phenomena of Spanish was presented in [Moreno, 1991], where some interesting and well founded linguistic generalizations are made: Paradigms[4] for verbs are described to capture regularities in the inflectional behaviour of the Spanish verbs, and the same is done with nouns.

[4]These are not the traditional ones, since they capture the problems arising in written language, such as diacritical marks, different surface letters for the same phoneme, etc.

All the *lemmas* belonging to a particular paradigmatic model not only share most of contextual and grammatical features but also have the same allomorph number and distribution. For instance, our model 11 has three allomorph stems, and their distribution is as follows:

```
vl(salir, 3, [0,12,13,14,15,16,21,22,23,24,25,26,
              31,32,33,34,35,36,61,62,63,64,65,66,
              82,85,90,99],[reg])                    --> [sal].
vl(salir,3,[11,51,52,53,54,55,56,83,86],[reg])       --> [salg].
vl(salir,3,[41,42,43,44,45,46,71,72,73,74,75,76],
           [fut_cond])                               --> [sald].
```

In [Goñi et. al., 1994] regular-expression based rules are devised to compute automatically these allomorphs, capturing morpho-graphemic generalizations in the paradigmatic models.

4 The grammar

The rule component of the model is quite small, because most of the information is in the lexicon. In particular, inflected verb forms are analysed or generated by two rules. Actually, only one rule is needed, but as we used the value 100 for the stt feature for regular verbs instead of a disjunction of all the possible stt values, we split the rule in two:

```
/**/
/*          VERB INFLECTION RULES          */
/**/
w(Lex, Cat, PerNum, TensMood) -->
            vl(Lex, Cat, Conj, SttL, SutL),
            vm(PerNum, TensMood, _,  ConjL, Stt, Sut),
        {
            member(Conj,ConjL),
            member(Stt, SttL),
            member(Sut,SutL)
        }.

w(Lex, Cat, PerNum, TensMood) -->
            vl(Lex, Cat, Conj, [100], SutL),
            vm(PerNum, TensMood, _, ConjL, _, Sut),
        {
            member(Conj,ConjL),
            member(Sut,SutL)
        }.
```


Nominal inflection is a bit more complicated, because of the combination of two inflectional morphemes (gender and number) in some cases. Our model needs the 4 rules shown to handle this. The first one is for singular words, when the stem has to be concatenated to a gender suffix (niñ-o, niñ-a); the second is for plural words, where an additional number suffix is added (niño-s); the third builds plurals from an allomorph stem and a plural morpheme (león / leon-es); and the fourth rule validates as words the singular forms (wl) obtained from the first rule without further concatenation:

```
/**/
/*      NOUN AND ADJECTIVE INFLECTION RULES    */
/**/

wl(Lex, Cat, [plu1], Gen, Num)
        --> nl(Lex, Cat, GetL, _, _, _), ng(Get, Gen, Num),
        {
                member(Get, GetL)
        }.

w(Lex, Cat, Gen, Num) -->
        wl(Lex, Cat, NutL, Gen, _), nn(Nut, Num),
        {
                member(Nut, NutL)
        }.

w(Lex, Cat, Gen, Num) -->
        nl(Lex, Cat, _, NutL, Gen, _), nn(Nut, Num),
        {
                Nut = plu2,
                member(Nut, NutL)
        }.

w(Lex, Cat, Gen, Num) -->
        wl(Lex, Cat, _, Gen, Num).
```

The predicate member included in the procedural part of the DCG rule implements disjunction in atomic contextual features, although it could have been eliminated with a different encoding of the lexical entries.

5 The Processor

The grammar rules are stated using the DCG formalism included in most Prolog implementations, thus we have used the DCG interpreter both for parsing and gen-

erating word forms. Since the interpreter is supplied with morphemes included in the dictionary for its proper operation, a segmenter has to be included to provide the parser with candidate word segmentations. This is achieved by means of a non-deterministic predicate that finds all the possible segmentations of a word. This is one of the efficiency drawbacks of the current implementation of GRAMPAL.

To avoid such inefficiency the system could be augmented with a *letter trie* index –or *trie*– [Aho et. al., 1983] to the lexical entries. With this device, segmentation will be no longer non-deterministically blind and the search would be efficiently guided. Generation does not have those efficiency problems, and the system is bidirectional without any change in the rules.

6 Conclusions

A Prolog prototype, GRAMPAL, was developed to intensively test the model, both as analyser and as generator. This processor implemented in Prolog has shown that logic programming can be used successfully to handle the Spanish inflectional morphology. We have also implemented a C version of GRAMPAL, but it needs separate components for analysis and generation, due to the lack of reversibility that Logic Programming has provided us with.

The model presented is based on two basic principles:

- Empirical rigour: all and only correct forms are analysed and generated, whether regular or not; gaps in verb paradigms are observed; suppletive forms are considered valid, and so on. It is important to stress that GRAMPAL does not overgenerate or overanalyse.
- Simplicity and generalization: GRAMPAL employs a really straightforward rule component, that captures the logical generalization of the combination of a stem and an ending to form a inflected word. "Standard scientific considerations such as simplicity and generality apply to grammars in much the same way as they do to any other theories about natural phenomena. Other things being equal, a grammar with seven rules is to be preferred to one with 93 rules" [Gazdar and Mellish, 1989].

The current dictionary has a considerable size: 43,000 *lemma* entries[5], including 24,400 nouns, 7,600 verbs, and 11,000 adjectives. The model could be used for derivative morphology and compounds as well, but this has not been done yet, since further linguistic analysis must be done to specify the features needed to permit derivatives and compounds.

[5]In these figures are neither included closed categories, nor allomorphs for verb and nominal morphemes.

References

[Aho et. al., 1983] Aho, A.V.; Hopcroft, J.E. and Ullman, J.D. (1983). Tries. *In Data Structures and Algorithms, ch. 5, sec. 3, pp. 163–169. Addison Wesley.*

[Antworth, 1990] Antworth, E. (1990). *PC-KIMMO: A Two-level Processor for Morphological Analysis.* Academic Computing Department, Summer Institute of Linguistics. Dallas, TX.

[Bear, 1986] Bear, J. (1986). A Morphological Recogniser with Syntactic and Phonological Rules. *Proceedings of the 11th International Conference on Computational Linguistics (COLING 86), pp. 272–276.*

[Gazdar and Mellish, 1989] Gazdar, F. and Mellish, C.S. (1989). *Natural Language Processing in Prolog.* Addison Wesley.

[Goñi et. al., 1994] Goñi, J.M.; González, J.C. and López, J. (1994). A Framework for Lexical Representation. *Technical Report UPM/DIT/LIA 5/94. Universidad Politécnica de Madrid, 1994.*

[Koskenniemi, 1985] Koskenniemi, K. (1985). A general two-level computational model for word-form recognition and production. *In Karlsson, F. (ed): Computational Morphosyntax, pp. 1–18. Dept. of Linguistics. University of Helsinki.*

[Martí, 1986] Martí, M.A. (1986). Un sistema de análisis morfológico por ordenador. *Procesamiento del Lenguaje Natural, n. 4, pp. 104–110.*

[Meya, 1986] Meya, M. (1986). Análisis morfológico como ayuda a la recuperación de información. *Procesamiento del Lenguaje Natural, n. 4, pp. 91–103.*

[Moreno, 1991] Moreno, A. (1991). Un modelo computacional basado en la unificación para el análisis y la generación de la morfología del español. *Tesis Doctoral. Universidad Autónoma de Madrid, 1992.*

[Ritchie et. al., 1987] Ritchie, G.; Pulman, S.; Black, A. and Russell, G. (1987). A Computational Framework for Lexical Description. *Computational Linguistics, vol. 13, n. 3–4, pp. 290–307.*

[Tzoukermann & Liberman, 1990] Tzoukermann, E. and Liberman, M. (1990). A Finite-State Morphological Processor for Spanish. *Proceedings of the 13th International Conference on Computational Linguistics (COLING 90), vol. 3, pp. 277–281.*

[Trost, 1990] Trost, H. (1990). The application of two-level morphology to non-concatenative German morphology. *Proceedings of the 13th International Conference on Computational Linguistics (COLING 90), vol.2, pp. 371–376.*

A Declarative Approach to the Design and Realization of Graphic Interfaces

D. Aquilino[♥], D. Apuzzo[♥], P. Asirelli[♦1]

Abstract

This paper presents the design and realization of a software Process model Editor for the OIKOS environment. This editor has been realized using Gedblog, a multi-theory deductive database management system. Gedblog allows applications, which heavily rely on graphic user interaction, to be developed and enacted according to a declarative style. It supports the consistent design and prototyping of graphic applications through an incremental development and/or by combining pre-defined theories. The case study we present highlights the Gedblog system features.

Keywords: Logic Programming, Graphical Interfaces, Database, Process Modelling.

1 Introduction

The main goal of this paper is to give some evidence that declarativeness supports graphic applications prototyping quite well. In this perspective, the Gedblog system [As94, As95a, As95b] has been used to realize the presented approach. Beside the outlining of new concepts and techniques provided by Gedblog, the main contribution relies in the realization of a complex, non-toy application by means of it. The application presented here is a graphic editor for OIKOS [Ap94] that allows to handle software process model schemata.

Gedblog is a deductive database management system for the design, validation and execution of interactive graphical applications. The main feature of the system is its declarativeness which allows users to develop their own applications in a compositional and consistent (with respect to the assessed requirements) fashion. This permits the graphical representation of concepts and their declarative semantics to be combined within a uniform, logic framework. An important feature of the system is the provision of an integrity constraints checking mechanism which permits to prove properties of the application under development. A graphic application is designed through a step by step refinement of the knowledge base. The application is developed consistently with respect to the defined integrity constraints that can be considered as the requirements that the application must satisfy.

The application knowledge is expressed as a deductive database [Sp84, Za90, We76] spread into multiple extended logic theories [Bro90, Mo89]. Each theory entails a piece of application data model and control by means of

- *facts* and *rules* to express the "general" knowledge;
- *integrity constraints* formulas to express either "exceptions" to the general knowledge, or general "requirements" of the application being developed;

1. Work partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of C.N.R. (National Research Council) and by project "Utilizzo di basi di dati deduttive per lo sviluppo software" founded by C.N.R. - Comitato Nazionale per le scienze e le tecnologie dell'Informazione (comitato 12)

♥ Intecs Sistemi S.p.A. - Via Gereschi, 32 Pisa; e-mail: {aquilino,mimmo}@sole.intecs.it

♦Istituto Elaborazione Informazione C.N.R. - Via S. Maria, 46 Pisa; e-mail: asirelli@iei.pi.cnr.it

- *transactions* to express atomic update operations that can be performed on the knowledge-base.

Graphic capabilities are handled by the system so that a graphic representation can be associated with a concept. The strategy is to use a logic language for the definition of graphic objects and their operations [He86, Hu86, Pe86, As87a, As87b, Pn90]. Since the representation is rigorously declarative, the graphic shape of an object is fully determined by the values of its attributes, and its visualization depends on finding a proof for the object in the database. The goal is to combine graphic and non-graphic components in the same declarative context. Graphic objects, their relationships, and the set of operations that the user can perform will be included in the knowledge of the application. The result of this integration will be that the overall semantics of the application can be affected by constraints imposed only on the graphic representation and, vice-versa, the graphic representation can be affected by semantics. Combining the concepts semantics and their graphic denotation within the same context it makes the tool particularly suitable for developing and prototyping applications in the areas of visual languages [Ch87, He90], graphic interfaces and some aspects of CAD.

2 The Gedblog System

The system is implemented following a client-server approach as it is shown in Figure 1. The client side realizes the user interface and thus it deals with the database editing capability. It requires a window management system to run (in the actual version we use Motif 1.2 [OS] on X11R5). However, Gedblog does not depend on any particular support. A different window system can quite easily be connected. The server is the part of the system that deals with the knowledge-base management. The actual version of the system is implemented in IC-Prolog [Co93] which forms the bottom layer. On this layer the Logic Kernel and the DBMS are built. They support knowledge handling at the logical level.

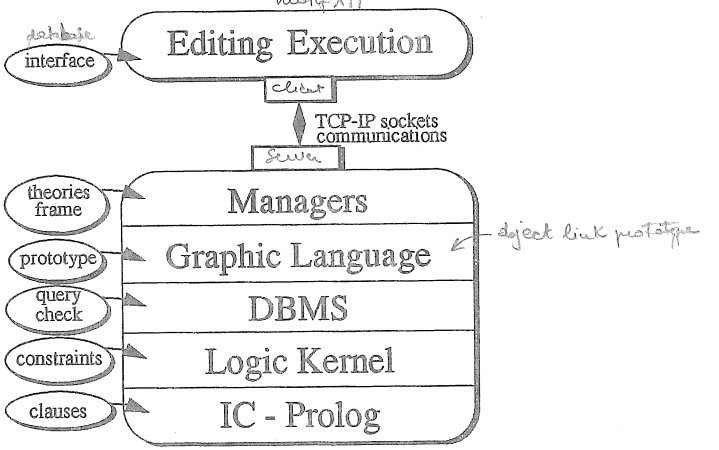

Figure 1: Gedblog Architecture

The DBMS is extended by the graphic component and by the managers. These make available to the clients the following features:

- multiple theories management, i.e. the capabilities to combine different pre-existing databases and load them as a single logical database;

- transactions management
- graphic objects management.

2.1 The Logic Kernel

In the following, some familiarity with Logic Programming and database terminology is assumed. Details can be found in [Ll87, Ga84].

The kernel of the system consists of a logic theory $T_{KB}=T_{DB} + T_{CONSTRAINT}$ where:

T_{DB} is a logic program (*facts*, i.e. the Extensional component of the DB (EDB) and *rules*, the Intensional component of the DB (IDB);

$T_{CONSTRAINT}$ are integrity constraint formulas of two kinds:

- *a set of Integrity Constraints (IC)*,

  $A_k >> B_1 \& ... \& B_s$

- *a set of Control formulas*

  $A_1 \& ... \& A_m ==> B_1 \& ... \& B_n$

The operator "&" stands for the logical connective $\wedge$.

Integrity checking is performed according to [As85]. The two forms of integrity formulas are used respectively to guarantee i) the construction of "correct with respect to constraints" applications (no illegal query will be answered) and ii) to perform controls at the user request.

2.2 The DBMS

The DBMS has been realized following a meta-programming approach [Sa86, St85]. It consists of the implementation of primitive updating operations plus an interpreter and a theorem prover (a Goal evaluator). The interpreter is used to evaluate user-defined updating operations (i.e. transactions).

The logic theory T_{KB}, described in the previous section is augmented by a meta-theory $T_{KBMS}= T_T + T_{PO}$ that realizes the management system.

T_T is the theory of transaction definitions: a set of clauses that define compound updating operations (*transactions*) with the following syntax and procedural interpretation:

$trans := \text{prec} \# t_1 \& ... \& t_n \# \text{post}$

T_{PO} is a set of elementary operations provided as a meta-theory with respect to the T_{KB}.

2.3 The Graphic Language

The graphic language layer supports a language for the definition and visualization of graphic objects thus enriching the theory T_{KBMS} by T_{GRAPH}. This theory contains the definition of graphic primitives, the rules needed to interpret the representations of graphic definitions and the meta-interpreter to visualize graphic objects.

Gedblog uses the technology supported by Motif [OS] and X11 [Sc86], and integrates it into a completely declarative environment. This means that "graphic objects" and "interactions" are specified by a language provided by Gedblog and based on a clausal representation according to the following elements:

- *prototypes*, i.e. templates of graphical objects;
- *graphic objects*, i.e. instances of prototypes;

2.4 The Server Managers

The managers layer defines the services that are provided to access, modify and load the knowledge stored into the server. These services are used by the clients to edit Gedblog databases and execute the resulting specifications.

Theory Manager

It allows to load multi-theories databases into the logic database server by means of the predefined predicate *theory*.

Graphic Frame Manager

The frame is the abstract logical space where objects are visualized and available for user interaction. In Gedblog the frame is managed in a strictly declarative way. This means that the objects on the frame at a certain instant are all and only those that can be proved (deduced) at that time in the knowledge base. This represent the most exclusive feature of Gedblog with respect other extensions of Logic Programming with graphics (such as many commercial Prolog implementations) and programming languages with graphical libraries in general.

2.5 The Gedblog client side

The Gedblog client supports the graphical interface for database editing and the execution of Gedblog applications. To do this, it interacts with the managers level of the server by means of a communication protocol based on messages exchange built on top of UNIX sockets.

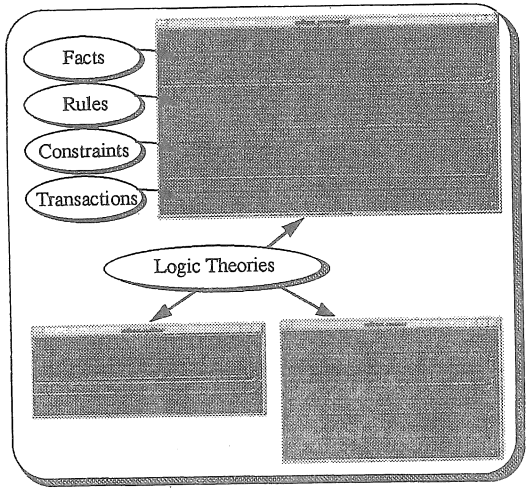

Figure 2: A Gedblog database

The database editor provides a graphical interface to edit Gedblog databases as a collection of multiple, heterogeneous theories. Each theory is visualized in a *View-Window*, divided into four *sections* respectively for the four different types of formulas.

Once a database is loaded into the server knowledge-base, the logic specification it defines can be executed. The Gedblog client implements the support to this phase. It shows the graphical objects that are stored in the logical frame and calls the transactions activated by the interaction of the user with the objects.

As transaction execution may update the knowledge-base, the client is also responsible to keep the visualized objects and their attributes up-to-date.

3 The OIKOS Process Editor

A very interesting and practical experiment in using Gedblog has been carried out by realising a graphical editor for OIKOS. OIKOS [Mo84] is an environment that provides a set of functionalities to easily construct process-centred software development environments. In the following a brief description of OIKOS is provided. This description will highlight the characteristics that have a direct impact with the realisation of the OIKOS editor.

3.1 Specification

In OIKOS a software process model is a set of hierarchical entities. Each entity is an instance of one of the OIKOS classes and represent a modelling concept. An entity can be either structured (i.e. formed by other entities) or simple (i.e. a leaf in the model structure). OIKOS defines a top-down method to construct process models. This method uses two descriptions of the entities: abstract and concrete. Abstract entities are introduced first and then refined into concrete ones. At the end of the modelling activity an enactable model is obtained by adding to the defined entities the needed details.

The method establishes some constraints about the entities (e.g. a coordinator entity has only a concrete representation) and their use as sub-entities during the model refinement (i.e. constraints in the model structure, e.g. a desk cannot have, among its sub-entity, a process). The entity classes defined in OIKOS are the following: Process, Office, Environment, Desk, Cluster, Session, Role and Coordinator. The OIKOS Process editor should provide the modeller with an environment to easily specify software process models following the OIKOS method. It should permit to edit and store different models for different users. Different modellers must operate with the editor with the same set of static constraints (i.e. the basic constraints of the method that cannot be changed because they are an integral part of the method) but they can work with different dynamic constraints (i.e. constraints on alternative ways to develop models). The editor interface layout is required to provide: menu for selecting operations, a top level entity to store the edited process, windows to present the informations of different entity kinds and dialogs for user inputs.

3.2 OIKOS Process Editor Database

This section describes the database that implements the OIKOS Process Editor [Ap94]. An overview of the theories that compose the OIKOS Editor database is given, with some explanations of the semantics they express. Theories have different purposes: to map the OIKOS model into logic predicates, to hold constraints on the modelling process, to define a graphical counterpart for OIKOS abstract entities, and to represent the editor layout using the suitable Gedblog mechanisms. From the user side, the theories are black-boxes. Users are just required to instantiate two theories, in order to load their personal instance of the editor database (oikos.editor theory) and to manage the data specific to the process they create during editor sessions (this theory will be referred to as oikos.<process_to_create>).

3.2.1 The Database Theories

The OIKOS Editor Database consists of the following theories: oikos.editor, oikos.model, oikos.menu, oikos.proto, oikos.<process_to_edit>

oikos.editor

This theory declares the imported theories and instantiates a window to display the root process. The theory is also used to record facts regarding the layout of the graphical frame

(exposed objects, selected objects, etc.). These facts are inserted by transactions along the course of execution sessions. The predicate *theory* is used to load all the theories that compose the OIKOS editor database, while the predicate *object* instantiates a window that stores the top-level entity of the process model.

```
theory('oikos.model').
theory('oikos.menu').
theory('oikos.proto').
theory('oikos.mini_dctu').
object(process_row,formDialog([(dialogTitle,string(top_level))],[],[
        link(the_row,rowColumn([(orientation,xmHORIZONTAL)],[],[]))])).
```

oikos.model

This theory introduces the predicates which define an abstract representation of OIKOS entities, and the constraints to be satisfied by OIKOS process structures.

```
kindc(process).kindc(office).kindc(env).kindc(desk).kindc(cluster).
kinda(ang_process).kinda(ang_office).kinda(ang_env).kinda(ang_desk).
kinda(ang_cluster).kinda(ang_role).kinda(ang_ses).
may(process,ang_desk).
conc(Name,Kind,Limbo) ==> kindc(Kind).
conc(Name,Kind,Angel,Res) ==> kinda(Kind).
part_of(Name1,Kind1,Name2,Kind2) ==> conc(Name1,Kind1,Limbo).
part_of(Name1,Kind1,Name2,Kind2) ==> may(Kind1,Kind2).
conc(Name,Kind,Limbo) ==> part_of(Name,Kind,Coord,coord).
```

The facts with predicates *kindc* and *kinda* distinguish angelic entity kinds from concrete ones respectively. A set of instances of predicate *may* is used to define the allowed inclusion relations among different kinds of entities. Then a set of static constraints is given to model OIKOS process models. The first and second constraints state that concrete and abstract entities must have respectively concrete and abstract kind. Notice the use of the *may* predicate in the fourth constraint formula. It states that an entity can be used as a part of another one only if the two entity kinds are in the *may* relation. Notice also that the theory defines only static constraints. However, dynamic constraints on the process construction methodology can be added by adding a separate theory.

oikos.menu

The menu theory builds the graphical object corresponding to the OIKOS Editor main menu and attaches transactions to the menu items in order to perform the corresponding actions.The menu-bar is realized by a fact in the theory that instantiate a menu-bar prototype, a set of facts that defines the prototypes for the pull-down items of the menu and the transactions to be performed in response to selection events.

```
object(main_dialog,formDialog([(dialogTitle,string(menu))],[],[
   link(oikos_menu,menuBar([],[],[ ...
prototype(obj_pull,pulldownMenu([],[],[
   link(coord,pushButton([(labelString,string(coord)),
      (activateCallback,callback(sel(coord)))],[],[])), ...])).
sel(Entity):= selected(Old) #out(selected(Old)) &in(selected(Entity))
     #true.
sel(Entity):= true#in(selected(Entity))#true.
```

The *sel* transaction is activated when an entity kind from the objects menu is selected. Its effect is to insert a new fact in the logic database (selected(Entity)) that records the entity kind that will be created next time an object create action is performed. The following picture shows the menu as it is visualized on the OIKOS Editor layout.

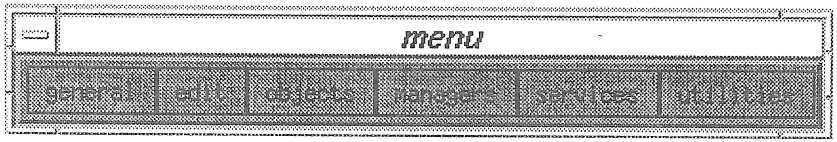

Figure 3: The Menu-bar

oikos.proto

Oikos.proto deals with the graphic presentation of entities. It declares the graphical prototypes, the rules upon which the entities visualization depends and the transactions to be activated in response to events occurring on them. According to the specification section, the entities that make up an OIKOS process description are: roles, coordinators, desks, clusters, environments, offices and processes. All but coordinators have an angelic counterpart. In order to attach a graphic representation to OIKOS Entities, consider they have a closed and opened status. When they are closed, only the icon representing the entity and the entity name are visible. The icons for entities and their angelic counterparts are given by the following Table:

	Process	Office	Env	Desk	Cluster	Role	Coord
Concrete							
Angelic							

Table 1: OIKOS Entities Icons

Closed entities are graphically represented by the following prototype definitions.

```
prototype(microf(Path,Kind,Name),form([(marginHeight,1)],[],[
   link(pb1,pushButton([(labelType,xmPIXMAP),labelPixmap,pixmap(Kind)),
      (activateCallback,callback(open(Name)))],[],[])),
   link(lb1,pushButton([(labelString,string(Name)),
      (activateCallback,callback(set_obj(Path,Name)))],[],[]))])).
```

This is a compound prototype that contains two buttons. One button shows the entity icon while the other one reports the entity name. Transactions are attached to button's activate events. The activate event on the icon button is linked to the *open* transaction by means of the callback mechanism.

As regards open entities, the graphical representation is different, depending on the following classification:

Compound Concrete Entities: Processes, Offices, Environments, Clusters and Desks are concrete compound entities. They are represented by a specification file written in the *Limbo* language and by the set of parts (sub-entities) that define their structure

```
prototype(generalconc(Name,Kind,Limbofile),
   formDialog([(dialogTitle,string(Name))],[],[
   link(fig,pushButton([(labelType,xmPIXMAP),
      (labelPixmap,pixmap(Kind)),
      (activateCallback,callback(close(Name)))],[],[])),
```



```
link(descr,pushButton([(labelString,string(Name)),
   (activateCallback,callback(set_active(Name,Kind)))],[],[])),
link(parts,partgrid(Name,Kind)),
link(limbolab,pushButton([(labelString,string(limbo_update)),
   (activateCallback, callback(save_file(Name!limbodef!limbodeftxt,
      'oikos.mini_dctu',Limbofile)))],[],[])),
link(limbodef,limbodefp(Name,Limbofile))
])).
```

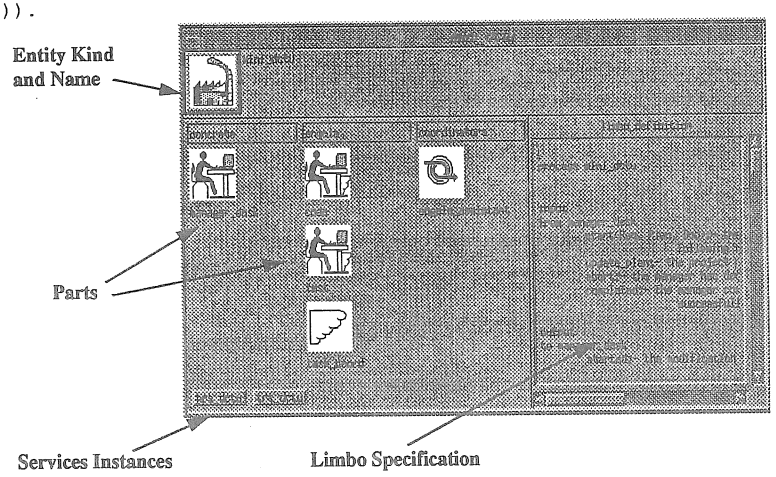

Figure 4: Compound Entity Instance

Angels: Angels represent the abstract specification of the entity. In Figure 9 an instance of an angelic entity is depicted.

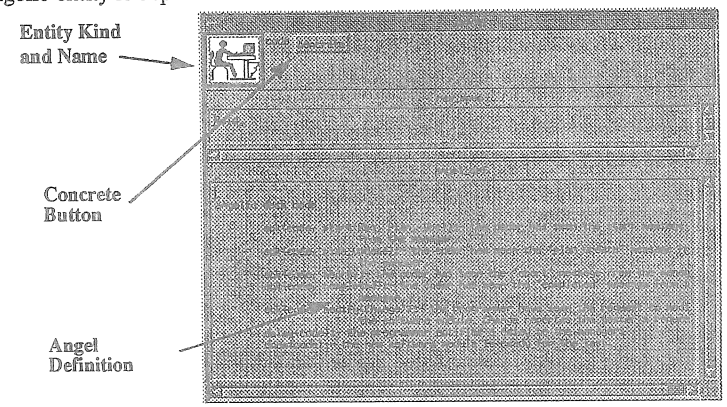

Figure 5: Angelic Entity Instance

Concrete Entities: Coordinators and Roles represent low-level entities in the OIKOS process description. Their definition is directly given into the entity of which they are parts.

The Oikos.proto theory connects also entities abstract representation to their graphical counterpart by the following rules:

```
object(Name,generalconc(Name, Kind, Limbofile)) <--
   conc(Name, Kind, Limbofile) & exposed(Name).
```

```
object(Name,generalabs(Name, Kind, AngSpec)) <--
   abs(Name, Kind, AngSpec) & exposed(Name).
link(Name1!parts!grid1,Name,microf(Name1!parts!grid1!Name,Kind,Name))<--
   part_of(Name1,Kind1,Name,Kind) & kindc(Kind) & exposed(Name1).
link(Name1!parts!grid2,Name,microf(Name1!parts!grid2!Name,Kind,Name))<--
   part_of(Name1,Kind1,Name,Kind) & kinda(Kind) & exposed(Name1).
```

The first group of rules states that a graphical object is on frame (see section 2.3.2 and 2.4) if the corresponding entity belongs to the knowledge base and the entity is in opened status (exposed). The second group of rules builds graphical objects corresponding to the parts of an open entity. Notice that these rules define the abstract data model of OIKOS process structure. A compound concrete entity is defined as an instance of the *conc* predicate. An angelic entity is modelled by the *abs* predicate, while the parts of a compound entity are described by the *part_of* predicate. The motivation for using two rules to put parts into the graphic grid of a compound entity is that we want to keep angelic and concrete sub-parts into different columns.

oikos.<process_to_edit>

This theory records the abstract representations of process entities. In general, this theory will be empty when the editor is started for the first time on a process. It will be enriched along to the user interaction with the editor.

The following example of this theory defines only the top level entity of a process model that is called *mini_dctu*.

```
top_level(mini_dctu).
conc(mini_dctu,process,mini_dctu).
part_of(mini_dctu,process,manager_desk,desk).
```

3.3 Execution of the OIKOS Editor Database

In the following execution simulation, we start the editor on the mini_dctu process as it is described by the theory Oikos.mini_dctu. At the start-up, the execution engine calls the frame manager to obtain the graphic objects to display. Recall that the frame manager returns the list of graphic objects that can be proved in the logic knowledge-base. The objects that will be visualized in this case are the menu bar and the process root icon., as they are facts of the database.

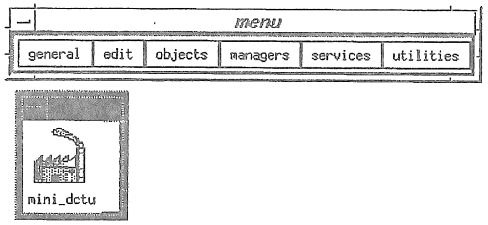

Figure 6: An editing session at start-up

Figure 6 shows the top-level representation of process *mini_dctu*. The process is represented by an icon, and is closed, in the sense that its internal definition is not visible. To open *mini_dctu*, the user clicks upon the icon button (the smoking factory). The *activate* event on the button calls the *open* transaction of the oikos.proto theory. This transaction inserts a new fact in the knowledge base: `exposed(mini_dctu)`. Now, look at the first rule of oikos.proto theory. It states that a compound concrete graphic object is deducible if the corresponding entity is defined in the database and it is exposed. So, exposing the mini_dctu entity causes

this rule to fire, as the fact `conc(mini_dctu, process, ...)` is true in the knowledge-base (refer to oikos.mini_dctu theory). The system pops-up the graphical representation of the mini_dctu entity as a compound concrete entity. The window shows the internal structure of the mini_dctu process entity. The only entity that appears in the mini_dctu structure is the manager desk. This is due to the fact `part_of(mini_dctu,process, manager_desk, desk)` in theory oikos.mini_dctu, and to rule 5 in theory oikos.proto. To insert a new entity among the mini_dctu parts, first select the entity on which to operate (in this case the mini_dctu process) by clicking its name. This event activate a transaction that inserts the fact `active(mini_dctu, process)` in the database.

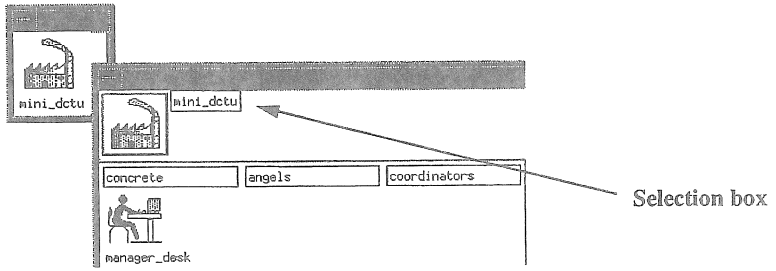

Figure 7: Selecting the active entity

Then chose the entity kind from menu *objects*. In the following example, a *coordinator* for the mini_dctu process is created by selecting the *coord* item from the *object* menu. A dialog window pops up, because the menu transaction asserts the fact `selected(coord)` and the rule

```
object(Kind,dialog(Entity,Etype,Kind))<--
   selected(Kind) & active(Entity,Etype).
```

is defined in the theory oikos.proto.

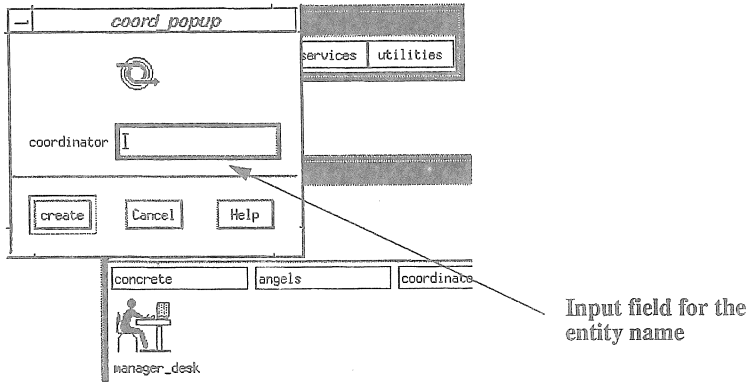

Figure 8: Creating a sub-entity

The dialog asks for the name of the new entity. After having filled up the name field with the name *new_coord*, the user clicks the create button. This action starts a transaction

```
mk_conc_entity(Mother,Kind,Type):=
   selected(Etype) & get_par(Type!mess!row1!nametxt,[(value,Name)]) &
   kindc(Etype)
   # out(selected(Etype)) & in(conc(Name,Type,Name) &
```

```
in(part_of(Mother,Kind,Name,Type))
# true.
```

that in this case instantiates the new facts `conc(new_coord,coord,Name)` and part_of(mini_dctu, process, new_coord, coord). Furthermore, the transaction removes the fact `selected(coord)`, so the dialog is dismissed. Now, the new coordinator is shown among the parts of the mini_dctu process by means of rules in oikos.proto theory. The correctness of the new schema is automatically granted by the constraints, which participate to the deduction process.

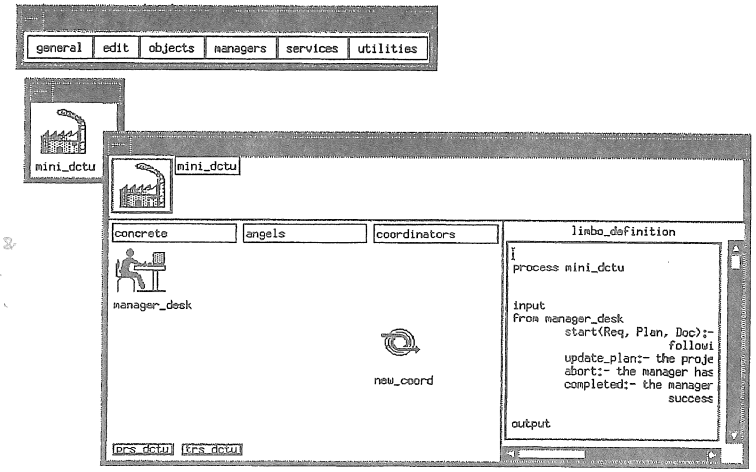

Figure 9: The Final Result

4 Conclusions

We have introduced Gedblog, and shown how a graphic application (the OIKOS process editor) has been implemented by a collection of Gedblog theories.

The (graphic) data language is an extension of the language used to define a deductive database. The syntax is based on Horn logic (definite clauses) and a mechanism is provided to handle integrity checking. Integrity constraints can be defined on graphic objects and on their visualization. Since graphic objects are handled within a logic database, all the advantages of logic are exploited so that the resulting system is declarative, deductive and its semantics is well founded. The final Gedblog system has been obtained by extending the logic database management system with a graphic language and mapping it to Motif and X11 primitives [Sc86]. At the present time, the system is implemented in C and IC-Prolog [Co93] and runs on a Sun 4 workstation under Unix 4.1.x [As90, As94b].

Besides Guided User Interfaces prototyping, one application area that seems to be very suitable for a system like Gedblog is the visual languages area [Ch80, Ch87], since it is very natural to assign, to each graphic object, its corresponding operational semantics that, usually, includes non graphic information as well.

As far as the applications implemented in Gedblog is concerned, the definition of the OIKOS process editor shows that only a very few rules are needed to define and handle something as complex as this application. The prototype is actually used within the OIKOS team, to develop process models for the OIKOS system.It constitutes a success, with respect other

attempts to implement the same graphical editor with standard programming languages and GUI support that failed.

Acknowledgements

We thank Prof. Carlo Montangero, for his help in the specification of the OIKOS process editor, and Dr. Chiara Renso, for her careful review of this paper.

References

[Ap94] D. Apuzzo, D. Aquilino, C. Montangero, OIKOS Process Editor Users Manual, Incecs Sistemi internal report, 1994.

[As85] P. Asirelli, M. De Santis, M. Martelli, Integrity Constraints in Logic Data Bases, Journal of Logic Programming, Vol. 2, No. 3, Ottobre 1985.

[As87a] P. Asirelli, P. Castorina, G. Dettori, A Proposal for a Graphic-Oriented Logic Database System, IEEE Proc. of The 2nd Int. Conf. on Computers and Applic., Pekin, June 1987.

[As87b] P. Asirelli, G. Mainetto, Integrating Logic DataBases and Graphics for CAD/CAM applications, IEEE WorkShop on Lang. for Automation,Vienna, August 1987, pp. 173 - 176

[As90] P. Asirelli, D. Di Grande, P. Inverardi, GRAPHEDBLOG Reference Manual, IR IEI B4-08 February 1990.

[As94] P. Asirelli, D. Di Grande, P. Inverardi, F. Nicodemi, Graphics by a Logic Database Management System, to appear in Journal of Visual Languages, 1995.

[As95a] P. Asirelli, P. Inverardi, D. Aquilino, D. Apuzzo, G. Bottone, M.C. Rossi, Gedblog Reference Manual(revised version), I.E.I. internal report B4 - 18, April 1995 and Progetto Finalizzato Sistemi Informatici e calcolo Parallelo Technical Report C - R/6/131, May 1995.

[As95b] P. Asirelli, D. Aquilino, G. Bottone, M.C. Rossi, Gedblog Users Guide (revised version), I.E.I. internal report B4 - 19, April 1995 and Progetto Finalizzato Sistemi Informatici e calcolo Parallelo Technical Report C - R/6/132, May 1995.

[Bro90] A. Brogi,P. Mancarella, D. Pedreschi, F. Turini, Composition Operators for Logic Theories, Computational Logic, Symposium Proceedings, editor J.W. Lloyd, Springer-Verlag, 1990.

[Ch80] N.S. Chang, K.S. Fu, A Relational Database System for Images, In: N.S. Chang and K.S. Fu (Ed.), Pictorial Information Systems, Springer, 1980, 288-321.

[Ch87] S-K Chang, Visual Languages: a tutorial and survey, IEEE Software 4, 1987, pp. 29-39

[Co93] Y. Cosmadopoulos, D. Chu, IC Prolog User's guide, *available by ftp from: rc.doc.ic.ac.uk in /computing/programming/languages/prolog/icprolog*.

[Ga84] H. Gallaire, J. Minker, J. M. Nicolas, Logic and Databases: a Deductive Approach, Computing Surveys, Vol.16, No.2, 1984, pp. 153 - 185.

[He86] R. Helm, K. Marriot, Declarative Graphics, Lecture Notes in Computer Science No. 225, Springer - Verlag, London, July 1986, pp. 513 - 527.

[He90] R. Helm, K. Marriot, Declarative Specification of Visual Languages, Proc. 1990 IEEE Workshop on Visual Languages, IEEE, pp. 98 - 103.

[Hu86] W. Hubner, Z. I. Markov, GKS Based Graphics Programming in Prolog, Computer Graphics Forum, Vol.5, March 1986, pp. 41 - 50.

[Ll87] Lloyd, J. Foundations of Logic Programming, 2nd edition, Springer-Verlag 1987.

[OS] OSF/Motif Programmers Guide, Open Software Foundation, Cambridge, MA.

[Mo84] C. Montangero, V. Ambriola, Oikos: Constructing Process-Centered SDEs, Software Process Modelling and Technology, editor A. Finkelstein and J. Kramer and B. Nuseibeh, Research Study Press distributed by J. Wiley and sons, London, 1994.

[Mo89] L. Monteiro and A. Porto, Contextual logic programming, "Proceedings Sixth International Conference on Logic Programming", G. Levi and M. Martelli (Eds.),The MIT Press, 1989.

[Pr90] M. J. Prospero, F. C. N. Pereira, On Programming an Interactive Graphical Application in Logic, Computer&Graphics Vol. 14, No. 1, pp. 7-16, 1990.

[Pe86] F. C. N. Pereira, Can Drawing Be Liberated from Von Neumann Style?, Logic Programming and Its Applications, M. van Caneghem e D. H. D. Warren Edd., A.P.C., Norwood, New Jersey, 1986, pp. 175 - 187.

[Sa86] S. Safra, E. Shapiro: Meta Interpreters for real, Inf. Proc. 86. H-J Kugler (Ed.), 1986, pp. 271-278.

[Sc86] R. W. Scheifler, J. Gettys, The X window system, ACM Transaction on Graphics, Vol.5, No.2, April 1986, pp. 79 - 109.

[Sp84] D. L. Spooner, Database Support for Interactive Computer Graphics, Proc. SIGMOD, 1984, pp. 90 - 99.

[St85] L. Sterling, Expert System = Knowledge + Meta-Interpreter, Dept. of Applied Mathematics, The Weizmann Institute of Science, Internal Report CS-84-17, 1985.

[We76] D. Weller, R, Williams, Graphics and Database Support for Problem Solving, ACM SIGGRAPH Computer Graphics, Vol.10, 1976, pp. 183 - 189.

[Za90] C. Zaniolo, Deductive Databases: Theory Meets Practice, (Invited Paper) Proc. EDBT'90, Lecture Notes in Computer Science No. 416, Springer - Verlag, pp. 1-15.

IMPLEMENTATIONS

Improving the Efficiency of Dynamic Modular Logic Languages

Anna Ciampolini[†], Evelina Lamma[†], Paola Mello[‡]
[†] *DEIS, Università di Bologna*
Viale Risorgimento 2, 40136 Bologna, Italy
[‡] *Dipartimento di Informatica, Università di Bari*
Via Orabona 4, 70126 Bari, Italy
{aciampolini,elamma,pmello}@deis.unibo.it

Abstract

In this paper, we focus on modular logic languages where module composition is performed through union of clauses and where the set of definitions to be used in order to evaluate a goal can be extended during the computation through implication goals. In order to improve the implementation of these languages, we discuss how to speed-up the search of (dynamic) bindings for predicate calls, and avoid this search for certainly failing computations. We consider the application of static analysis techniques and an implementation based on the Warren Abstract Machine. In particular, we apply abstract interpretation in order to compute the minimal sets of modules (*minimal contexts*) which are needed for any successful computation for a given predicate. The proposed implementation exploits a bit vector representation for the current set of modules and bit-wise operations in order to speed-up the search for predicate binding. The results of the analysis are exploited by the implementation in order to reduce the overhead due to failing, unuseful computations.

1 Introduction

Several applications both in the Software Engineering and the Artificial Intelligence area require some forms of modularity and structuring of knowledge. This has motivated a considerable research effort about *modular logic programming* which has led to several proposals and implemented systems (see [5] for a survey).

In this paper, we show how to improve the efficiency of modular logic languages. In particular, we focus on modular languages having implication goals of the kind $D \supset G$ (where D is a set of possibly extended clauses, and G is a goal) in the body of clauses. The set of definitions for goal evaluation can be extended during the computation through implication goals, and the resulting language adopts *dynamic scope rules*. In this respect, the current set of definitions is the union of the clauses belonging to the modules involved in the implication goals encountered. Thus, clauses can see each others, whichever the order of execution of implication goals has been.

In the case of modular languages adopting dynamic scope rules, the binding of predicate calls to definitions has to be performed only at run-time, when the call is raised. The binding requires to perform a dynamic search in the run-time program representation.

Therefore, a key point for a successful implementation is to reduce the overhead due to this search. This can be obtained in two ways. First, by avoiding the search when possible. Second, by enhancing the efficiency of the search technique adopted. In this paper, we face both the approaches, and we show how they can co-exist in a single implementation, based on the Warren Abstract Machine.

With respect to search avoiding, we consider the application of static analysis techniques [7] in order to avoid the overhead due to try to bind a predicate which certainly leads to failing, unuseful computations. In particular, we use abstract interpretation to determine the *minimal contexts* (i.e., the minimal sets of modules) which are needed in order to have a possibly successful computation for a given predicate p. When a call to p is raised, if the current set of modules in use is not a superset of any of the minimal contexts of p, then the computation fails immediately. If no minimal context exists, then the call can be replaced by a failure through a proper program transformation. The impact of this kind of analysis is particularly relevant when an *explicit* representation of the set of modules currently in use is maintained by the underlying implementation.

With respect to enhancing the efficiency of the search, existing implementations for this class of languages basically maintain a run-time representation of the program in terms of the list of its component clauses. Whenever an implication goal $D \supset G$ is encountered, the clauses in D are added to the list, and they are discardedas soon as G is deterministically solved or finitely fails.

Most of the implemented systems (see for example [10, 9]) rely on a run-time program representation defined in terms of internal data structures manipulated by the underlying engine. All these implementations have some overhead due to the dynamic search of predicate definitions. In fact, in [10] a linear search takes place dynamically for each predicate call, whereas in [9] a new hash function has to be computed for each implication goal.

In this paper, we propose an implementation where, at run-time, the current set of modules in use (*context*, for short) is represented as a bit vector. An implication goal involving module m_i just amounts to setting to 1 the i-th bit of this bit vector. Moreover, a bit vector is associated at compile time to each predicate p in order to record the set of modules (and addresses of clauses) defining p. When a call to p is raised, efficient bit-wise operations are used for inspecting the current set of modules in order to find predicate bindings. Either linear search or the computation of a new hash function are no longer needed.

The implementation also exploits the results of the abstract analysis in order to reduce the overhead due to failing, unuseful computations. In particular, the set of minimal contexts of each predicate p is maintained by the underlying implementation. When a call to p is raised, before determining the binding for it, the system checks if the current set of modules in use is a superset of (or is equal to) at least one *minimal context*. If this is not the case, the call will certainly lead to a failure, and thus the binding of the predicate call is not even searched for.

2 The Modular Language and its Semantics

In this section, we give a formal description of the modular logic language in term of both operational and fixpoint semantics.

2.1 Syntax and Operational Semantics

The modular language we consider is a proper extension of Horn clause logic: it is in practice that defined by Miller in [11], provided that no explicit quantification occurs for implication goals. A program is composed by a set of modules. Each module consists of a set of (possibly extended) definite clauses, and is identified by a unique constant name. Clauses can contain implication goals of the kind $D \supset G$ in their body, where D is a module and G is a goal..

In his seminal paper [11], Miller he formalized the operational semantics of implication goals by extending the provability relation $\vdash$ for Horn Clauses with the following inference rule:

$$\text{(AUGMENT)} \quad \frac{\mathcal{P} \cup D \vdash G}{\mathcal{P} \vdash D \supset G}$$

In the following discussion, we will assume that modules are designated by names, and that the same name may have multiple occurrences in a program. In particular, modules can be introduced as named collections of clauses and programs can be structured as collections of modules each one dedicated to answer a specific class of queries. Cross-referencing between modules and module-composition can then be accounted for relying on the workings of implication goals. If, in module M, the answer to a goal G requires that the clauses of module M_1 be loaded, then we will simply enforce the evaluation of G in the composition of M and M_1 by means of the implication goal $M_1 \supset G$.

For reasons of notational convenience, we will often adopt the conventional Prolog syntax for clauses and write implication goals as M >> G where M is a module name. In doing so, we will also assume that a clause $A{:-}G_1, ..., G_n$ is in normal form, i.e., that all the variables occurring in it are universally quantified.

Example 2.1 *Let us consider the following program:*

```
m1: delete_or_append(X,Y,Z):-m4>>r(X,Y,Z).
m2: delete(X,[],[]).
    delete(X,[X|Y], Z):-delete(X,Y,Z).
m3: append([],X,X).
m4: r([X],Y,Z):-delete(X,Y,Z).
    r(X,Y,Z):-append(X,Y,Z).
    delete(X, [U|Y], [U|Z]):-delete(X,Y,Z).
    append([X|U],V,[X|W]):-append(U,V,W).
```

Knowledge about list manipulation predicates is split into several modules. Module `m1` *defines predicate* `delete_or_append/3` *which causes, in practice, through predicate* `r/3` *defined in module* `m4`*, a non-deterministic selection between the* `append` *operation (defined in modules* `m3` *and* `m4`*) and the* `delete` *operation (defined in modules* `m2` *and* `m4`*).*
In this program, the goal `m1>>m3>>delete_or_append([g],[a,b,c,d,e,f],X)` *succeeds with answer* `X/[g,a,b,c,d,e,f]`.

Other modular languages with the same structure, but adopting more strict scoping rules have been proposed by many authors (e.g., [8, 12, 4]). For a survey refer to [5].

2.2 Fixpoint Semantics

In [3], the authors provide a model-theoretic and fixpoint semantics for the modular language here presented. The two declarative semantics are proved equivalent to the operational one. In this section, we recall the fixpoint semantics since it will be adopted as a basis for the abstract analysis (see section 3).

Let P be a program and $\mathcal{M}_P$ the set of names of the modules of P. The Herbrand base of P, $\mathcal{B}_P$, is defined as usual [14] as the set of all the ground atomic formulas built from the Herbrand universe of P. The Herbrand universe is the standard one, and we suppose it is disjoint from the set of module names $\mathcal{M}_P$.

Definition 2.2 *Let P be a program. The set of program compositions* (contexts) *which can be built from the modules of P is defined as: $\mathcal{C}_P = Powerset(\mathcal{M}_P)$.*

Being $\mathcal{M}_P$ finite, $\mathcal{C}_P$ is finite as well.

Definition 2.3 *Let P be a program. The* extended Herbrand base *is defined as:*

$$\mathcal{B}_{CP} = \{\langle c, A\rangle \mid c \in \mathcal{C}_P, A \in \mathcal{B}_P\}$$

An interpretation is any subset of $\mathcal{B}_{CP}$.

Thus, the interpretation domain is the powerset of $\mathcal{B}_{CP}$, and it is a complete lattice under set inclusion.

Definition 2.4 *Given a context c, a goal formula G and an interpretation I, we denote by $c \models_I G$ the fact that G is true in c with respect to I.*

The immediate consequence operator is defined in terms of the following truth relation.

Definition 2.5 *Let P be a program, $I \subseteq Powerset(\mathcal{B}_{CP})$, $A \in \mathcal{B}_P$, $M \in \mathcal{M}_P$ and G, G_1, G_2 goal formulas, and $c \in \mathcal{C}_P$. The truth relation $\models_I$ is defined as follows:*

$c \models_I \top$
$c \models_I A \quad iff \quad \langle c, A\rangle \in I$
$c \models_I G_1, G_2 \quad iff \quad c \models_I G_1 \ \wedge \ c \models_I G_2$
$c \models_I M \supset G \quad iff \quad \{M\} \cup c \models_I G$

Definition 2.6 *Let P be a program. The transformation function*

$$T_P : \ Powerset(\mathcal{B}_{CP}) \ \mapsto \ Powerset(\mathcal{B}_{CP})$$

is defined as follows:

$$T_P(I) = \{\langle \{m\} \cup c, A\rangle \mid \exists\ G \supset A \in ground(m) \ \wedge \ m \in \mathcal{M}_P \ \wedge \ c \models_I G\}$$

where $ground(m)$ represents the grounded version of m's code.

As proved in [3], this transformation function is monotone and continue, and therefore has a least fixpoint.

3 The Abstract Analysis

In this section, we show how abstract interpretation techniques can help in improving the implementation of dynamic modular logic programs.

The aim is to statically determine, for each predicate symbol p occurring in a given program P, the *minimal contexts* (i.e., the minimal sets of modules) which are needed in order to possibly have a successful computation. More precisely, if a call to a predicate p is raised in a set of modules C (i.e., a *context*) which is not a superset of (or equal to) any minimal context of p, then the computation for p will certainly lead to failure, and therefore it is convenient to raise a failure immediately. Notice that the fact that C is equal or is a superset of some minimal context of p does not guarantee that a successful computation for p exists. In this respect, the condition above is only a *necessary* but not a *sufficient* condition.

The analysis of modular logic languages has also been considered in [6]. In that work, Codish *et al.* develop an abstract interpretation for the compositional semantics proposed in [2] for *open logic programs*. In the modular language considered in [6], however, a program consists of a set of modules composed by *definite clauses* only. Thus, no linguistic extension is introduced, and module composition can be seen as a meta-linguistic approach (see [5] for discussion). This allows to also obtain the nice property of having a compositional analysis: altering one module does not require to re-analyse the entire program.

Our approach is different in two respects. First, we consider a richer class of languages, also encompassing dynamic module composition. For these languages no compositional semantics has been developed till now, and thus the analysis we develop is not compositional as well. Furthermore, we develop an abstract interpretation of a fixpoint semantics with the specific purpose of improving efficiency by avoiding expensive, unsuccessful computations.

3.1 The Abstract Domain and Transformation Function

For the concrete domain, $\mathcal{D}$, we rely upon the definitions introduced in section 2.2. In particular, $\mathcal{D}$ is defined as the powerset of the extended Herbrand base (see definition 2.3): $\mathcal{D} = Powerset(\mathcal{B}_{CP})$

Definition 3.1 *Let $\mathcal{P}_P$ denote the set of predicate symbols occurring in program P. We define $\mathcal{D}^\alpha = Powerset(\mathcal{C}_P \times \mathcal{P}_P)$*

We associate with the domain $\mathcal{D}^\alpha$ a pre-order relation, $\sqsubseteq_{pre}$, defined as follows.

Definition 3.2 *Let I^α and J^α be two elements of $\mathcal{D}^\alpha$. We say that $I^\alpha \sqsubseteq_{pre} J^\alpha$ iff $\forall\langle c, p\rangle \in I^\alpha\ \exists\langle c', p\rangle \in J^\alpha : c' \subseteq c$.*

To get a partial order, it suffices to consider the domain $\mathcal{D}^\alpha$ modulo the equivalence relation induced by the pre-order $\sqsubseteq_{pre}$ by grouping together those sets that are equivalent with respect to $\sqsubseteq_{pre}$ [13]. Two sets I and J belong to the same equivalent class iff $I \sqsubseteq_{pre} J$ and $J \sqsubseteq_{pre} I$.

Let $[\![\mathcal{D}^\alpha]\!]$ denote the quotient of $\mathcal{D}^\alpha$ with respect to $\sqsubseteq_{pre}$. The equivalence classes are *partially ordered* by $\sqsubseteq_{pre}$: for equivalence classes $P, Q \in [\![\mathcal{D}^\alpha]\!]$, $P \sqsubseteq Q$ iff for all $A \in P$ and $B \in Q$, $A \sqsubseteq_{pre} B$. Let $[I^\alpha]$ represent the equivalence class containing the set $I^\alpha \in \mathcal{D}^\alpha$.

We choose $[\![\mathcal{D}^\alpha]\!]$ as abstract domain. It can be proved that $[\![\mathcal{D}^\alpha]\!]$ equipped with the partial order $\sqsubseteq$ is a complete lattice, with bottom element $\bot^\alpha = [\emptyset]$, and top element $\top^\alpha = [\{\langle \mathcal{C}_P, p\rangle \mid p \in \mathcal{P}_P\}]$.

Abstraction is performed by the function α, defined as $\alpha = Pred \circ Min \circ \{\_\}$ where:

- $Pred : \mathcal{D} \mapsto \mathcal{D}^\alpha$ is such that $\forall I \in \mathcal{D}$:

$$Pred(I) = \{\langle c, p\rangle \mid \langle c, A\rangle \in I \ \wedge \ p \text{ is the predicate symbol of } A\}$$

- $Min : \mathcal{D}^\alpha \mapsto \mathcal{D}^\alpha$ is such that $\forall I^\alpha \in \mathcal{D}^\alpha$:

$$Min(I^\alpha) = \{\langle c, p\rangle \mid \nexists \langle c', p\rangle \in I^\alpha : c' \subset c\}$$

- $\{\_\} : \mathcal{D}^\alpha \mapsto [\![\mathcal{D}^\alpha]\!]$ maps $I^\alpha \in \mathcal{D}^\alpha$ to $[I^\alpha]$

In practice, the function $Pred$ abstracts each atom $A \in \mathcal{B}_P$ with its predicate symbol p. As concerns function Min, a set of contexts, $C(q)$, can be associated with each predicate symbol q appearing in a given interpretation $I^\alpha \in \mathcal{D}^\alpha$:

$$C(q) = \{c \mid c \in \mathcal{C}_P \wedge \langle c, q\rangle \in I^\alpha\}$$

The powerset of $C(q)$ is a complete partial order: each chain in this set represents the contexts associated to q linked together by the (usual) subset relation $\subset$. Function Min reduces each set $C(q)$ of I^α to the set composed by only the lower bound elements of each chain. In other words, the function Min eliminates from the set I^α all couples $\langle c, p\rangle$ such that $c \supset c' \wedge \langle c', p\rangle \in I^\alpha$. For this reason, each context c appearing in $Min(I^\alpha)$ is said to be *minimal*. Finally, function $\{\_\}$ maps elements of $\mathcal{D}^\alpha$ in equivalence classes of $[\![\mathcal{D}^\alpha]\!]$.

Let us now introduce the concretization function:

Definition 3.3 *The concretization function $\gamma : [\![\mathcal{D}^\alpha]\!] \mapsto \mathcal{D}$ is defined as follows:*

$$\forall [I^\alpha] \in [\![\mathcal{D}^\alpha]\!] : \gamma([I^\alpha]) = \{\langle c, A\rangle \mid \langle c, p\rangle \in I \wedge I \in [I^\alpha] \wedge p = pred(A) \wedge A \in \mathcal{B}_P\}$$

Therefore, for an equivalence class $[\{\langle c', p\rangle\}] \in [\![\mathcal{D}^\alpha]\!]$, function γ generates all the couples $\langle c, A\rangle \in \mathcal{D}$ such that $A \in \mathcal{B}_P$ have predicate symbol p, and $c' \subseteq c$. In practice, besides mapping predicate names into ground atoms, γ builds a set of chains having c' as the lower bound, thus generating all possible contexts including c'.

Assuming the ordering relationship $\subseteq$ on the concrete domain $\mathcal{D}$ and the ordering relationship $\sqsubseteq$ on the abstract domain $[\![\mathcal{D}^\alpha]\!]$, it can be proved that the abstraction function α is total and monotone. Furthermore, the pair of functions α and γ yields a Galois insertion [7].

Proposition 3.4 *Given $(\mathcal{D}, \subseteq)$ and $([\![\mathcal{D}^\alpha]\!], \sqsubseteq)$, the following properties hold:*
(i) α is total and monotone;
(ii) γ is total and monotone;
(iii) $\forall d \in \mathcal{D} : \ d \subseteq \gamma(\alpha(d))$;
(iv) $\forall d^\alpha \in [\![\mathcal{D}^\alpha]\!] : \ \alpha(\gamma(d^\alpha)) = d^\alpha$. □

Relying upon definition 2.6, we define the abstract transformation function, T^α_P, as the composition: $T^\alpha_P = \alpha \circ T_P \circ \gamma$ As usual, the abstract analysis of the given program P should iteratively apply the transformation function T^α_P, starting from the set $\bot^\alpha = [\emptyset]$, until the least fixpoint is reached. In practice, the least fixpoint is a set of couples $\{\langle c, p\rangle\}$ where each element represents a *minimal context* (c) that might (but doesn't guarantee) lead to a successful computation for an atom A such that: $pred(A) = p$. The advantage is that we can statically determine which contexts will certainly lead to failure for A (i.e., all contexts that do not include any context belonging to couples in $T^\alpha_P \uparrow \omega$), and thus we can avoid to execute certainly unuseful computations.

Example 3.5 *Let us consider the program of example 2.1. We are interested in finding, for each predicate symbol of $\mathcal{P}_P$, the minimal contexts which are separately needed for a possibly successful computation, through the iterated application of the abstract function, as follows:*
$T^\alpha_P \uparrow 0 = \bot^\alpha = [\emptyset]$
$T^\alpha_P \uparrow 1 = \alpha(T_P(\gamma(\bot^\alpha))) = \{\langle\{m2\}, delete\rangle, \langle\{m3\}, append\rangle\}$
$T^\alpha_P \uparrow 2 = T^\alpha_P \uparrow 1 \cup \{\langle\{m2, m4\}, r\rangle, \langle\{m3, m4\}, r\rangle\}$
$T^\alpha_P \uparrow 3 = T^\alpha_P \uparrow 2 \cup \{\langle\{m1, m2\}, delete\_or\_append\rangle, \langle\{m1, m3\}, delete\_or\_append\rangle\}$
Finally, in the next iteration T^α_P reaches its least fixpoint:
$T^\alpha_P \uparrow 4 = T^\alpha_P \uparrow 3 = T^\alpha_P \uparrow \omega$
Thus, from the results of the abstract analysis we can conclude that any call to predicate `delete_or_append` *certainly fails if neither* `m1` *and* `m2` *nor* `m1` *and* `m3` *belong to the current context.*

4 The Implementation

The implementation of a modular logic language based on implication goals must take into account the dynamic evolution of the structure of a program, and thus support the dynamic update of the bindings for a predicate call. Accordingly, the implementation must provide adequate data structures for the run-time representation of a program.

The treatment of implication goals conceptually amounts to "asserting" and "retracting" program clauses. Since implication goals can be nested arbitrarily, several nested applications of these operations may have to be performed at run-time. However, the assertion and retraction of program clauses follows a stacking discipline, and may be as such implemented using a run-time stack. Of course, backtracking will need special treatment as it may require the reinstatement of a program "asserted" at earlier stages of the computation.

A natural way to think about the run-time representation of a program is in terms of the list of its component clauses. Whenever an implication goal $D \supset G$ is encountered, the clauses in D are added to the list, and as soon as G is deterministically solved or finitely fails, they are discarded.

In the following, we present an implementation for the language introduced in section 2. This implementation extends the Warren's Abstract Machine (defined by D.H.D. Warren in [15]) with new instructions and data structures, and exploits the results of the abstract analysis described in section 3. The implementation provides separate compilation of each module. Albeit, this feature is not fully expoited by the propesed analysis since this is not

compositional. Throughout this section, we assume familiarity with the workings of the WAM.

4.1 The Bit Vector Implementation

Each module of a program P is separately compiled. Moreover, for each predicate p, the compiler produces the following structures, maintained in the code area of the program: (1) a *predicate vector*, i.e. a bit vector recording the modules where predicate p is defined. If p has a definition in module m_i, for instance, then the i-th bit of the predicate vector of p is set to 1; (2) a *predicate frame*. This data structure records, for each module m_i defining p, the address of the first instruction for p in m_i's compiled code.

Since module composition corresponds to union of clauses (see the $AUGMENT$ rule in section 2), the implementation maintains the dynamic representation of a program (i.e., the set of modules currently in use or *context*) through a *bit vector* (with dimension equal to the number of modules of the program). At each instant of the computation, the run-time representation of the set of clauses in use is, in practice, given by the current context. In terms of the bit vector representation, if module m_i belongs to the current context, the i-th bit is set to 1. In the underlying extended WAM, a new register `CR` maintains the current context as a bit vector[1]. `CR` register is possibly modified when an implication goal is encountered. The new WAM instructions `augment` and `shrink` are added for handling implication goals. For instace, the implication goal $m>>p(t)$ is compiled into the following code:

```
augment  m
...          /* put of arguments for goal p */
call p/1
shrink
```

The `augment` instruction has the module name (say m_i) as argument, and sets to 1 the corresponding bit (say i-th) of `CR`. The previous value of `CR` is copied in a continuation register (`CCR`), saved in the environment. The `shrink` instruction restores the previous value of `CR` from `CCR` register.

Furthermore, the content of `CR` is saved in the choice point, and restored upon backtracking.

As concerns the binding of predicate calls, in the standard WAM the address used by the `call` and the `execute` instructions is statically determined. In the implementation of dynamic modular languages, instead, this address must be determined dynamically. Determining the bindings between predicate calls and definitions represents the major source of run-time overhead for these systems as the cost of binding resolution amounts to one look-up access in the program representation to determine the correct address for the call.

Thanks to the bit vector organization, the cost for binding predicate calls can be notably reduced. In fact, binding a predicate call for p requires only to perform intersection operations between `CR` register and the predicate vector of p, and then to access the proper code in the predicate frame of p. To this purpose, in the implementation two registers

[1]The current implementation supports a program with at most 32 modules (32 bits for `CR`) but it can be straightforwardly extended to cope with a larger number of modules.

are added to the WAM, representing the predicate vector (`PV`), and the predicate frame address (`PA`). Both these registers are saved in the choice point[2]. When a `call` instruction is raised for predicate p, the predicate vector of p is loaded in `PV` register, and a reference to the predicate frame of p is loaded in `PA` register. If the bit-wise intersection between `PV` and `CR` is not null, then, for each bit set to one (say i-th), the corresponding i-th slot of the predicate frame is accessed in order to get the address of procedure p of the i-th module of the program. Then, the i-th bit of `PV` is reset to 0. It is worth to notice that, since modules are accessed in order of position within the `PV` vector, we rely on an implicit ordering among them.

Since each predicate can be defined in several units, we have to handle inter-module non-determinism. To this purpose, before accessing the code for p through the predicate frame, a choice point is always allocated, even if p is deterministic. In this way, the information about the set of definitions not yet used but to be considered for a predicate (represented by the bit-wise intersection of `PV` and `CR` and by the value of `PA`) is saved in the choice point, and restored when intra-module backtracking fails. In order to maintain a single data structure for handling both inter- and intra- module backtracking, `try` and `try_me_else` instructions set `PV` register to *nil*.

The bit vector implementation also avoids to allocate several times the same module in the program representation. This is, instead, a problem in other implementations where the treatment of implication goals may result in the same module activation being added several times to the program representation (e.g., when we have recursion through an implication goal). This has a potential drawback since it may cause useless searching to be performed, and the same solution to be produced several times. In our implementation, instead, the number of copies of any module in a program representation is restricted to just one. In fact, the program representation is interpreted as the union of (the clauses of) component modules. Hence, multiple occurrences of the same module are safely avoided.

4.2 The Impact of the Analysis

The abstract analysis presented in section 3 yields for each predicate p the set of minimal contexts that are necessary for a successful computation for p. This information can be employed during the computation to avoid to raise a call when it will certainly fail.

In the following, we refer to the implementation schema described in 4.1. This schema is further extended by considering the results produced by the abstract analysis. More in detail, we associate with each predicate p of the program its *minimal contexts*. We call this set the *label* of p. Each *minimal context* in the label of p is represented by a bit vector. For instance, if a module m_i belongs to a *minimal context* c, the i-th bit in the vector representing c is set. With this representation schema, a call to a predicate p implies a dynamic check of the current context. If the current context in use is a superset of (or is equal to) a particular *minimal context*, the call *might* lead to a successful computation; otherwise (i.e., none of the minimal contexts of predicate p is a subset of - or equal to - the current context), the call will certainly lead to a failure and therefore can be safely avoided.

Each comparison between the current context (represented by the content of `CR` register) and one of the element of the *label* of predicate p can be easily implemented by

[2]The bit-wise intersection between `CR` and `PV` is saved in the choice point.

bit-wise logic operations between bit vectors. In fact, being $v_{p,j}$ the bit vector associated with the j-th *minimal context* c_j of p, the check is performed by the sequence of two logic operations: (1) one bit-wise *AND* operation between `CR` and $v_{p,j}$; the result (r) represents the intersection between the context and the *minimal context* c_j; (2) one bit-wise *EXOR* operation between $v_{p,j}$ and r. If the result is the null vector (i.e., all zeros) the current context is a superset of (or equal to) the minimal context, and the call to p is raised since it might lead to success. No further check has to be done.

In case of a not null result, `CR` has to be compared with the remaining *minimal contexts*. Only if all checks give a not null result the call to predicate p will be discarded.

The overhead imposed by this modification is related to the number of minimal contexts returned by the abstract analysis for each predicate p. In particular, the number of comparisons between bit vectors is less or equal to the number of *minimal contexts* associated with p. However, the cost of each comparison is very limited, due to the very low cost of bit-wise operations. Moreover, it is worth to notice that the highest overhead is generated: (1) in the case of a failing call, since `CR` has to be compared with all minimal contexts; or (2) when only the last *minimal context* is a superset of `CR`. In the first case, the overhead is largely balanced by the advantage deriving from avoiding the call; the second case, instead, is made less probable by ordering the *minimal contexts* in decreasing order of size.

In the following example, we estimate the benefits deriving from the use of this modified version of the implementation schema.

Example 4.1 *Let us consider the program of example 2.1 and its analisys shown in example 3.5:*

$$T_P^{\alpha} \uparrow \omega = \{\langle\{m2\}, delete\rangle, \langle\{m3\}, append\rangle, \langle\{m2, m4\}, r\rangle, \langle\{m3, m4\}, r\rangle,$$

$$\langle\{m1, m2\}, delete\_or\_append\rangle, \langle\{m1, m3\}, delete\_or\_append\rangle\}$$

The set of minimal contexts obtained for this program can be represented by the following predicate labels*:*
$L(delete\_or\_append) = \{[1,1,0,0],[1,0,1,0]\}$
$L(delete) = \{[0,1,0,0]\}$
$L(append) = \{[0,0,1,0]\}$
$L(r) = \{[0,1,0,1],[0,0,1,1]\}$
Let us consider the top goal `:- m1>>m3>>delete_or_append([g],[a,b,c,d,e,f], X)`. *In this case, the use of abstract analysis can avoid to raise an expensive call to the predicate* `delete`. *In fact, when the call to* `r` *is raised, the first clause in* `m4` *is activated. After the unification, the abstract machine compares the value of* `CR` *($[1,0,1,1]$) with the* minimal contexts *in label $L(delete)$ in order to test if* `CR` *is a superset of (or equal to) at least one them. The check gives an unsuccessful result, and the call to* `delete` *is therefore avoided. It is worth to notice that without this check, the call to* `delete` *predicate would have caused a number of applications of the* `delete` *clause in* `m4` *equal to the length of the second argument of* `delete_or_append` *(i.e., equal to six in the example above), before reaching a failure condition (due to the non visibility of module* `m2`*).*

In the example above, the abstract machine globally executes, in the worst case, seven comparisons between `CR` and predicate *minimal contexts*; as shown before, each check is performed with a couple of bit-wise operations. Thus, the additional load due to these checks is the one generated by 14 bit-wise operations between vectors. Therefore, the overhead imposed in the run-time support, when exploiting abstract analysis results, is much lower than the gain of efficiency due to the avoidance of six `delete` recursive calls in `m4`, plus the consequent backtracking.

Moreover, thanks to the analysis, in some cases we can further reduce the overhead due to the run-time checks. In particular, as soon as a call to a predicate p that has just one minimal context associated with is raised, the check is performed for p but not for subsequent subgoals.

With respect to existing implementations of modular systems with dynamic scope rules [10, 9], the bit vector implementation substantially improves both the search of predicate bindings and the treatment of implication goals. Either linear search (performed in [10]) or the creation of a new access function for predicate binding (performed in [9]) are no longer needed, and predicate binding is simply obtained by bit-wise operations.

We have compared the performance of the bit vector implementation with that of the system described in [10]. Detailed results are reported in [1]. For the `nreverse` program, when partitioned into several modules, the overhead due to the predicate binding and the treatment of implication goals is reduced of about fifty percent if the bit vector implementation is adopted instead of that in [10].

5 Conclusions

In this paper, we have focused on modular logic languages with implication goals and dynamic scope rules, and discussed on how to improve their implementation by using static analysis techniques and a bit vector representation. In particular, we have applied abstract interpretation in order to reduce the overhead due to failing, unuseful computations. Moreover, the bit vector representation reduces the overhead due to the dynamic search of predicate bindings and the handling of implication goals. The bit-vector implementation can exploit quite directly the results of the analysis: the resulting abstract machine provides a non-neglectable reduction of the overhead due to failing calls because of the absence of some needed module in the current context. The implementation is an extension of the Warren Abstract Machine. The first prototype has been written in C, and runs on different platforms.

Future work will be devoted to test both the implementation and the impact of the analysis on more significant examples. It is worth to notice, however, that the abstract analysis results could be similarly used to improve other implementations such as those described in [10, 9]. The impact of the analysis on these systems will be a subject for future work. Moreover, we plan to investigate how to extend the analysis in order to reduce backtracking. In practice, we would like to introduce a form of indexing on program clauses in order be able to find the right clause to apply on the basis of the current context. To this purpose, we need a more sophisticated analysis that is able to distinguish between different clauses defining the same predicate and therefore associates mimimal contexts with them.

Acknowledgements

We would like to thank Francesco Battini for its help in implementing the extended WAM and the referees for their useful comments and remarks on the previous version of this paper. This work has been partially supported by C.N.R. "Project on Program Analysis and Transformation ANATRA".

References

[1] F. Battini, E. Lamma, and P. Mello. The Implementation of a Logic Language with Dynamic Modules. Technical report, Department of Computer Science, University of Bologna, 1993.

[2] A. Bossi, M. Gabbrielli, G. Levi, and M.C. Meo. Contributions to the semantics of open logic programs. In H. Tanaka, editor, *Proc. FGCS'92 Int. Conference*, pages 570–580, 1992.

[3] A. Brogi, E. Lamma, and P. Mello. Hypothetical Reasoning in Logic Programming: A Semantics Approach. *Information Processing Letters*, 36:285–291, 1990.

[4] M. Bugliesi. A Declarative View of Inheritance in Logic Programming. In K. Apt, editor, *Proc. Joint Int. Conference and Symposium on Logic Programming*, pages 113–130. The MIT Press, 1992.

[5] M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19-20:443–502. 1994.

[6] M. Codish, S.K. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *Proc. 20th Annual ACM Symp. on Principles of Programming Languages*, pages 451–464. ACM Press. 1993.

[7] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13:103–180, 1992.

[8] L. Giordano, A. Martelli, and G.F. Rossi. Local definitions with static scope rules in logic languages. In *Proc. FGCS'88 Int. Conference*, pages 389–396, 1988.

[9] B. Jayaraman and G. Nadathur. Implementation techniques for scoping constructs in logic programming. In K. Furukawa, editor, *Proc. 8th Int. Conference on Logic Programming*, pages 871–886. The MIT Press, 1991.

[10] E. Lamma, P. Mello, and A. Natali. An Extended Warren Abstract Machine for the Execution of Structured Logic Programs. *Journal of Logic Programming*, 14:187–222, 1992.

[11] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[12] L. Monteiro and A. Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Proc. 6th Int. Conference on Logic Programming*, pages 284–302. The MIT Press, 1989.

[13] D.A. Schmidt. *Denotational Semantics*. Allyn and Bacon Inc., 1986.

[14] 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.

[15] D.H.D. Warren. An abstract Prolog instruction set. Technical Report TR 309, SRI International, 1983.

Lazy Narrowing on an Abstract Machine by Means of Examples*

Eva Ullán-Hernández

Dpto. de Informática y Automática
Universidad Complutense de Madrid
Avda. Complutense s/n, 28040 Madrid, Spain
phone: +34 1 394 44 20, fax: +34 1 394 46 07
email: evah@eucmvx.sim.ucm.es

Abstract

This paper outlines an abstract machine design for a lazy functional logic language. Our aim is to integrate into an existing narrowing machine ([6, 9, 10, 12]) the *demand driven strategy* for lazy narrowing presented in [11]. The underlying narrowing machine is a combination of a particular reduction machine with mechanisms for unification and backtracking based on the Warren's Abstract Machine ([1, 15]). We have modified and extended it in order to fulfil our purposes. Trying to illustrate what is to be done, we focus the presentation on a running example.

Keywords: Abstract machines, functional logic languages, lazy narrowing, demand driven strategy.

1 Introduction

Functional and logic programming are the most important declarative programming paradigms, and its integration has been extensively investigated during the last decade. The operational semantics of (lazy) functional logic languages is based on (lazy) narrowing. Informally, narrowing is a natural extension of reduction to incorporate unification. For a survey on the development of the operational semantics as well as on the implementation methods of functional logic languages, see [7].

A common difficulty within all implementations of lazy narrowing is to find good computation strategies, which avoid repeated evaluations of arguments and "minimize" the risk of non-termination. A *naive* computation strategy will lead to many repeated evaluations of arguments. Additionally, its interaction with backtracking may lead to non-termination, as it has been shown in [6]. A proposal that tends to

*This research has been supported by the Spanish National Project TIC92–0793 "PDR".