8.1 A Parallel Execution Model

The Incrementality Lemma 7.1 for $\mathcal{F}$ suggests a possible parallel execution model of clp's, based on a network of processors:

Network Let N be the set of pp's of $\mathcal{P}$. For $l \in N$, a *processor* P_l is associated with l.

Communication among processors is realized by means of channels, as follows:

Communication Processors are connected by the following *channels*: (i) $c_\omega^{entry(G)}$ from the environment ω to $P_{entry(G)}$ and $c_{exit(G)}^{\omega}$ from $P_{exit(G)}$ to the environment; (ii) c_j^i from j to i for every i, j such that there is an arc from j to i in $dg(\mathcal{P})$.

A channel c_j^i is called an *input channel of* P_i and an *output channel of* P_j. Each channel is supposed to have a *memory* that contains a queue of states whose policy is fair (e.g. first in first out).

The execution model allows the processors to run in parallel and asynchronously:

Execution Model Processors in the network execute asynchronously the following *algorithms*:
- $P_{entry(G)}$ takes an α from $c_\omega^{entry(G)}$ and sends it to all its output channels.
- $P_{entry(C)}$ selects with *fair choice* from one of its input channels, say $c_{call(A)}^{entry(C)}$, an α, and it computes $\beta = (push(\alpha) \wedge \bar{s}^1 = \bar{t}^0)$, where $A = p(\bar{s})$ and $p(\bar{t})$ is the head of H; then $P_{entry(C)}$ sends β to every its output channel.
- $P_{success(A)}$, where A is not a constraint and is contained in the clause C, selects with fair choice from one of its input channels, say $c_{exit(D)}^{success(A)}$, an α; then it computes $\beta = pop(\alpha)$; if $\beta \in \neg free(x_C^0)$ then $P_{success(A)}$ sends β to every its output channel.
- $P_{success(A)}$, where A is a constraint, takes an α from its input channel and computes $\beta = (\alpha \wedge A^0)$, then $P_{success(A)}$ sends β to every its output channel.

This model describes a sound and complete implementation of $\mathcal{O}$, as stated in the following theorem.

Theorem 8.1 (Adequacy of $\mathcal{M}$) *If the input channel c_ω^e of $\mathcal{M}$ is feed with the set of states ϕ s.t. $\phi \subseteq \neg free(x_G^0)$, and $\phi \subseteq free(x_C^0)$ for every non goal, non-unitary clause C, then $\bigcup_{\pi \in path(l)} psp.\pi.\phi$ is the set of states that P_l in $\mathcal{M}$ sends on its output channels.*

Remark 8.2 Our execution model assigns one processor to each program point. However, because the processors work asynchronously, in case there are less processors than program points, then a single processor can be assigned to a number of pp's, which can be encoded as distinct tasks to be executed with a fair schedule discipline. This will still yield a complete and asynchronous model.

8.2 Burstall's Intermittent Assertions Method

We show how the intermittent assertions method of Burstall [Bur74] can be adapted to clp's. The advantages of the Intermittent Assertion Method, and of Temporal Logic (TL) in general, for instance to prove *liveness properties*, *termination*, *total correctness* etc. are well known (see for instance [CC93]). So far, finding a suitable presentation of the intermittent assertion method for logic programming was still an open problem ([CC93]). In this section a solution to this problem for clp's is given, by means of the intermediate semantics $\mathcal{O}$.

For lack of space, we shall be rather sketchy and we refer the interested reader to the full version of the paper.

For simplicity assertions are denoted by ϕ, ψ, thus identifying an assertion with the set of states it denotes. Implication is interpreted as set inclusion, i.e. $\phi \Rightarrow \psi$ iff $\phi \subseteq \psi$. Also, conjunction and disjunction are interpreted set-theoretically as intersection and union, respectively. The assertion $push(\phi)$ is obtained by replacing each i-variable x^i in ϕ by the i-variable x^{i+1}; and $pop(\phi)$ is

obtained by first renaming with fresh variables all the i-variables of level 0 and then replacing each remaining i-variable x^i with x^{i-1}.

Here, an 'intermittent rule' is a formula in temporal logic of the form $\Box(\phi \wedge \mathrm{at}(i) \Rightarrow \Diamond(\psi \wedge \mathrm{at}(j)))$, where $\Box$ and $\Diamond$ are the 'always' and 'sometime' operators, and $\mathrm{at}(i)$ indicates that execution is at program point i. The set of proof rules we consider contains a formalization of the induction principle (Burstall's "little induction"), a suitable axiomatization of TL (cf. [Sti92, CC93]), plus the following *path rule*, which formalizes the "hand simulation" part of the method:

$$(\pi \in path(i,j) \wedge psp.\pi.\phi \neq false) \Rightarrow \Box(\phi \wedge \mathrm{at}(i) \Rightarrow \Diamond(psp.\pi.\phi \wedge \mathrm{at}(j)))$$

A *sound* and *relatively complete* proof system w.r.t. $\mathcal{F}$ can be defined using these tools.

We illustrate by means of an example how the method can be applyed to prove total correctness of a clp. The following *composition rule* will be used:

$$\frac{\Box(\phi \wedge \mathrm{at}(i) \Rightarrow \Diamond(\psi \wedge \mathrm{at}(j))) \qquad \Box(\psi \wedge \mathrm{at}(j) \Rightarrow \Diamond(\chi \wedge \mathrm{at}(k)))}{\Box(\phi \wedge \mathrm{at}(i) \Rightarrow \Diamond(\chi \wedge \mathrm{at}(k)))} \tag{1}$$

It enables us to compose intermittent assertions (note this is a particular case of the 'chain rule' which is one of the basic tools in the proof system presented in [MP83]).

Example 8.3 Consider again the program *Prod*. Let the initial assertion be $\phi = u^0 = [r_0, \ldots, r_k] \wedge \neg free(x_G^0) \wedge free(x_{C1}^0) \wedge \mathrm{at}(1)$.

Suppose we want to prove that *Prod* satisfies the following assertion:

$$\Box(\phi \Rightarrow \Diamond(v^0 = r_0 * \ldots * r_k \wedge \mathrm{at}(2))) \tag{2}$$

which says that for every state α of ϕ, at least one execution of $\leftarrow prod(u, v)$ starting in α terminates (i.e. reaches the pp 2) and its final state binds v to $r_0 * \ldots * r_k$. Below we use A, B as a shorthand for '$A \wedge B$' (i.e. comma stands for conjunction). Using the path rule we get the following (simplified) assertions:

$\Box(\phi \Rightarrow \Diamond(v^1 = z^0 = r_0 * w^0, y^0 = [r_1, \ldots, r_k]), \mathrm{at}(4)))$ (with path $\langle 1, 3, 4 \rangle$)

$\Box(v^{k+1} = z^k = r_0 * \ldots * r_k * w^0, y^0 = [], \mathrm{at}(4) \Rightarrow \Diamond(v^{k+1} = z^k = r_0 * \ldots * r_k, y^0 = [], \mathrm{at}(5)))$ (with path $\langle 4, 6, 5 \rangle$)

$\Box(v^1 = z^0 = r_0 * \ldots * r_k, \mathrm{at}(5) \Rightarrow \Diamond(v^0 = r_0 * \ldots * r_k, \mathrm{at}(2)))$ (with path $\langle 5, 2 \rangle$)

The following assertions can be proven by straightforward induction:

$\Box(v^{m+1} = z^m = r_0 * \ldots * r_m * w^0, y^0 = [r_{m+1}, \ldots, r_k], m < k, \mathrm{at}(4) \Rightarrow \Diamond(v^{k+1} = z^k = r_0 * \ldots * r_k * w^0, y^0 = [], \mathrm{at}(4)))$
(using as path $\pi = \langle 4, 3, 4 \rangle$), and

$\Box(v^{k+1} = z^k = r_0 * \ldots * r_k, y^0 = [], \mathrm{at}(5) \Rightarrow \Diamond(v^1 = z^0 = r_0 * \ldots * r_k, \mathrm{at}(5)))$ (using as path $\pi = \langle 5, 5 \rangle$)

Then, the repeated application of rule (1) to compose the above assertions yields (2). □

9 Discussion

In this paper an alternative operational model for clp's was proposed, where a program is viewed as a dataflow graph and a predicate transformer semantics transforms a set of states associated with a fixed node of the graph (corresponding to the entry-point of the program) into a tuple of set of states, one for each node of the graph. To the best of our knowledge, this is the first predicate transformer semantics for clp's based on dataflow graphs. The dataflow graph provides a static description of the flow of control of a program, where sets of constraints 'travel' through its arcs. The relevance of this approach was substantiated in the Applications section.

We would like to conclude this paper by giving an extension of its results to more general CLP systems. We have considered 'ideal' CLP systems. With slight modifications, the dataflow semantics

$\mathcal{F}$ (and all its applications) can be adapted to deal also with 'quick-check' and 'progressive' systems (cf. [JM94]), which are those more widely implemented. This can be done as follows. States are considered to be pairs (c_1, c_2) of constraints, instead than constraints, where c_1 denotes the active part and c_2 the passive part.

$$States = \{(c_1, c_2) \mid c_1 \text{ and } c_2 \text{ are constraints s.t. } consistent(c_1)\},$$

where the test $consistent(c_1)$ checks for (an approximation of) the consistency of c_1. Then rules **R** and **C** of Table 1 have to be changed as illustrated below, where a state $\alpha = (c_1, c_2)$ is also denoted by (α_1, α_2):

$$\mathbf{R} \quad (\langle p(\overline{s})\rangle \cdot \overline{A},\ \alpha\ ) \longrightarrow (\ \overline{B} \cdot \langle pop\rangle \cdot \overline{A},\ infer(\alpha'_1, \alpha'_2 \wedge \overline{s}^1 = \overline{t}^0)\ ),$$

with $\alpha' = push(\alpha)$, if $C = p(\overline{t}) \leftarrow \overline{B}$ is in $\mathcal{P}$.

$$\mathbf{C} \quad (\langle d\rangle \cdot \overline{A},\ \alpha\ ) \longrightarrow (\ \overline{A},\ infer(\alpha_1, \alpha_2 \wedge d^0)\ ),$$

if d is a constraint. Finally, the definition of sp has to be changed in:

$$sp.c.\varphi = \{\alpha' \in States \mid \alpha' = infer(\alpha_1, \alpha_2 \wedge c) \text{ and } \alpha \in \varphi\}.$$

The operator *infer* computes from the current state (c_1, c_2) a new active constraint c'_1 and passive constraint c'_2, with the requirement that $c_1 \wedge c_2$ and $c'_1 \wedge c'_2$ are equivalent constraints. The intuition is that c_1 is used to obtain from c_2 more active constraints; then c_2 is simplified to c'_2. For instance, in the example of Section 5, in the state of Ψ_4^2 the constraint $z^0 = x^0 * w^0$ would be passive, because the equation is not linear (cf. [JMSY92]). Then, in Ψ_6^3 this constraint is transformed by applying first *push* to it and then *infer* . So $z^1 = x^1 * w^1$ becomes active, because w^1 is bound to 1 and hence the equation becomes linear.

Acknowledgments: We would like to thank Jan Rutten and the anonymous referees for their useful comments.

References

[BGLM94] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The *s*-semantics approach: theory and applications. *The Journal of Logic Programming*, 19,29: 149–197, 1994.

[Bur74] R.M. Burstall. Program proving as hand simulation with a little induction. *Information Processing*, 74:308–312, 1974.

[CC93] P. Cousot and R. Cousot. "A la Burstall" Intermittent Assertions Induction Principles for Proving Inevitability Properties of Programs. *Theoretical Computer Science*, 120:123–155. 1993.

[CMM95] L. Colussi, E. Marchiori and M. Marchiori. On Termination of Constraint Logic Programs. In *Proc. First International Conference on Principles and Practice of Constraint Programming*. LNCS, Springer–Verlag, 1995. To appear.

[JMSY92] J. Jaffar, S. Michaylov, P.J. Stuckey and R.H.C. Yap. The CLP($\mathcal{R}$) Language and System. *ACM TOPLAS*, 14(3):339–395. 1992.

[JM94] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *JLP* 19,20: 503-581, 1994.

[Mel87] C. Mellish. Abstract interpretation of Prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of declarative languages*, pp. 181–198. Ellis Horwood, 1987.

[MP83] Z. Manna and A. Pnueli. How to cook a proof system for your pet language. In *Proceedings 10th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 141–154, 1983.

[Nil90] U. Nilsson. Systematic semantics approximations of logic programs. In *Proc. PLILP*, pp. 293–306. Eds. P. Deransart and J. Małuszyński, Springer Verlag, 1990.

[Sti92] C. Stirling. Modal and Temporal Logics. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, vol. 2, pp. 477–563, 1992.

[YI94] S. Yamasaki and K. Iida. Transformation of Logic Programs to FP Programs Based on Dataflows. *Journal of Symbolic Computation*, 17-18:157–182, 1994.

[WA84] W.W. Wadge and E.A. Ashcroft. LUCID, the dataflow programming language. Academic Press, London, 1985.

Labeling in CLP(*FD*) with evolutionary programming.

Alvaro Ruiz-Andino Illera, José J. Ruz Ortiz

Dpto. Informática y Automática
Fac. CC. Matemáticas. Universidad Complutense de Madrid
Av. Complutense s/n. Madrid 28040 SPAIN
Tel: 34 - 1- 394 44 68 - Fax: 34 -1 - 394 46 07
e-mail: {arai,jjruz}@dia.ucm.es

Abstract: Constraint logic programming over finite integer domains allows a declarative and flexible statement of combinatorial optimization problems. The paradigm used is "constraint and generate". Constraints prune in advance the search space, and then, a enumeration phase, also called labeling, and a search strategy are needed to find the optimal solution. In this paper we introduce the integration of evolutionary algorithms, a well known computing paradigm, and the CLP(*FD*) paradigm. We have designed a system that enhances CLP with techniques based in evolutionary programming, allowing to solve a constrained optimization problems without the need of programming an explicit and exhaustive enumeration and specifying a strategy to find the optimal solution. The paper describes the algorithms used to implement the evolutionary program, and also the design details of the genetic operators. Finally, we present an example of the operation of the prototype of the system, which has been implemented in ECLiPSe.

Keywords: CLP(*FD*), constrained optimization problems, evolutionary programming.

1 Introduction.

The main advantages of applying the CLP(*FD*) approach (Constraint Logic Programming over finite integer domains) [VH89] to cost optimization problems are its flexibility and ease of programming [Din90,Wal94]. Combinatorial optimization problems over natural numbers are defined as follows: given a set of variables ranging over natural numbers, a set of constraints between these variables, and an objective function, the problem is to find an assignment of values to the variables that satisfies the constraints and optimizes (i.e., minimizes or maximizes) the objective function.

The basic paradigm used to solve this kind of problems in CLP is "constraint and generate", but in many cases after the constraint phase the remaining search space can be quite large, so the way labeling is performed plays an important role. CLP supports search over a solution space structured into a tree, some of whose leaves are feasible solutions. The constraints allow to prune in advance during the search some of the branches whose leaves include no feasible solutions. Optimization problems require not just a feasible solution but an optimal one, assuming some function associating a cost with each solution. Finding the optimum requires some kind of enumeration of the feasible solutions. The enumeration efficiency can be improved using problem specific heuristics and/or general methods like branch and bound. But the size of many constrained search problems prevent the problem from being tackled by any complete search technique, even when the search space may be pruned by constraint handling. For such problems approximation algorithms

are a good alternative. These kinds of algorithms do not guarantee to find an optimal solution, but offer a high probability of finding a good solution by exploring only a part of the solution space. In this paper we introduce the integration of an approximation technique called evolutionary programming into CLP(*FD*) to solve constrained optimization problems. This extension allows the programmer to be freed from specifying the labeling and optimization strategies.

1.1 Background on evolutionary programming.

An evolutionary program is a stochastic computational device, based on principles of evolution and hereditary, that allow an effective search in very large search space. Evolutionary programming comes from the refinement of genetic algorithms. As hold in [Mic94], "genetic algorithms + data structures = evolution programs". For many hard search problems, such as the traveling salesman problem, assembly-line sequencing and scheduling, evolutionary algorithms have been used very successfully [Gol89]. The skeleton of an evolutionary program in shown in Figure 1.

```
procedure evolutionary program
begin
   t := 0;
   initialize P(t)
   evaluate P(t)
   while  not termination-condition  do begin
      t := t+1;
      select P(t) from P(t-1)
      alter P(t)
      evaluate P(t)
   end;
end.
```

Figure 1.

An evolutionary program maintains a population of "chromosomes", $P(t) = \{X_1^t, ..., X_n^t\}$ for iteration t. Each chromosome represents a potential solution to the problem at hand, implemented as some, possibly complex, data structure S. The initial population $P(0)$ is generated randomly or by any other method (`initialize` step). Then the population is evaluated (`evaluate` step), computing a "fitness" value for each chromosome X_i^t that indicates a measure of the goodness of the chromosome as a solution to the optimization problem. The objective function to be optimized is the basis for the computation of this fitness value. Then a new population, $P(t+1)$, is formed by selecting some chromosomes (`select` step) from $P(t)$. Best fitted solutions are more likely to be chosen for survival. Some members of the new population undergo transformations by means of "genetic" operators to form new solutions (`alter` step). There are unary transformations m_i (mutation type), which create new solutions by a small change in a single chromosome (m_i: $S \rightarrow S$), and transformations c_j (crossover type), which create new solutions by combining two (or more) randomly selected chromosomes (c_j: $S \times S \rightarrow S$). Best fitted solutions are more likely to be chosen for crossing-over. After some number of generations (iterations) the program converges — it is hoped that the best chromosome represents a near-optimum solution. There is a theoretical foundation for this kind of algorithms based in the schemata theorem [Hol75], which is beyond the scope of this introduction.

Section 2 introduces the details of the of the evolutionary algorithm designed for its integration in CLP(*FD*) in order to solve constrained optimization problems. Section 3 presents an example and some empirical results. Finally we discuss the conclusions and future work.

2 Labeling with evolutionary programming.

A constrained optimization problem within the framework of CLP(*FD*) may be stated in the following way: given a tuple $<V, D, C, f>$, where:

- $V \equiv \{V_1, ..., V_n\}$: finite set of domain variables.
- $D \equiv \{D_1, ..., D_n\}$: finite set of integer domains associated to the variables V_i.
- $C \equiv \{C_1, ..., C_m\}$: finite set of constraints between the variables in V.
- f: objective function ranging over V.

find an assignment of values from D to the variables in V that satisfies the constraints in C and optimizes (maximizes or minimizes) the objective function f. First, constraints C_i are stated leading a reduction of the original domains, and then a labeling strategy is needed to perform the search for the optimal assignment. Our aim is to enhance CLP(*FD*) with an optimization technique that, given a list of domain variables and a cost expression, returns a near optimal solution with respect to the cost expression. Searching will be performed using a evolutionary constrained algorithm, using constraints to guide the genetic operators to a feasible solution. In this section we introduce the main points of the design of the evolutionary program.

2.1 Representation of solutions.

In highly constrained problems, a minimal change to a feasible solution is very likely to generate an unfeasible one, but unfeasible solutions cannot simply be dropped from the solution space because doing so would prevent certain good solutions from being generated. Classical approaches overcome this problem using one or more tricks like penalty functions, the avoidance of generating illegal solutions, repair algorithms, linear recombination [Min92,Mic93]. The integration of evolutionary algorithms with the constraint propagation and local consistency techniques embedded in CLP over finite integer domains offer a new approach to solve this problem. We introduce an approach where chromosomes do not represent a "ground" solution, but an "area" of the search space, that is, variables are not labeled with an integer value, but a integer domain, so a chromosome may include none or many solutions. Local consistency [VH92] and constraint propagation does not guarantee that a not completely ground chromosome includes a solution, but it may contain many, both good and bad, covering an area of the search space. Genetic operators have been designed in order to both guarantee the convergence to a "ground" solution while exploring as much as possible the search space. During generation and recombination of chromosomes local consistency and constraint propagation is triggered, keeping chromosomes within the feasible solution space as much as possible.

A chromosome X_i, which represents a set of potential solutions, is formed by the list $[d_1, .., d_n]$ where each d_i is an abstract data type representing the integer domain associated with the variable i of the list of domain variables to be labeled. Each

chromosome, a potential solution, has an associated cost d_c, which is also the integer domain associated to the cost expression to optimize.

Besides the lists of d_i's, we will also keep some extra information about each chromosome for the implementation of the evolutionary mechanisms: its fitness value (`fit`), its accumulated relative fitness (`arf`), used in the `select_chromosome` step, and a boolean flag to indicate if it will survive to the next generation.

2.2 The algorithm.

Figure 2 shows the main algorithm implemented in our system. It clearly follows the skeleton shown in Figure 1. The following subsections will describe each step in detail, giving the algorithm used for each underlined step.

```
procedure  evolutionary_labeling_in_CLP (Vars : list of fd_vars;
                                         f : objective function);
begin
  t := 0;
  initialize P(t):
    for i:=1 to pop_size do
      X_i := random_labeling(Vars);
  evaluate P(t):
    for  i:=1 to pop_size  do
      X_i.fit := fitness(f,X_i)
    total_fitness := Σ X_i.fit
    X_i.arf := Σ (j :=1 to i) X_j.fit / total_fitness
  while  not termination-condition  do
    select P(t) from P(t-1):
      for  i:=1 to pop_size * prop_surv  do  begin
        X_j := select_chromosome(P(t))
        mark X_j to survive
      end;
    alter P(t):
      for  i:=1 to pop_size  do
        if  not X_i marked for survival  then begin
          X_1 := select_chromosome(P(t))
          X_2 := select_chromosome(P(t))
          replace X_i with crossover(Vars,X_1,X_2)
        end;
      for  i:=1 to pop_size  do
        X_i := mutation(Vars,X_i)
    evaluate P(t)
  end;
  final_solution := best_solution(Vars,P(t))
end.
```

Figure 2.

The initialize step is a loop that generates *pop_size* chromosomes by means of a `random_labeling` procedure. Evaluate step computes the `fitness` value of each chromosome of the population, and also its accumulated relative fitness. In the select step, some randomly chosen chromosomes are marked as survivors, so they won't be replaced by the new chromosomes generated by crossover. Best fitted chromosomes are more likely to be selected, as `select_chromosome` procedure make a random selection based in the accumulated relative fitness. The `alter` step has been divided in the two genetic

operators, `crossover` and `mutation`. New chromosomes generated by crossover take the place in the population of those chromosomes that were not chosen for survival. New chromosomes generated by means of mutation replace the chromosome used to produce the mutation.

2.3 Generating the initial population.

The first step in any evolutionary program is the generation of an initial population. Each chromosome $X_i^0 = [d_1, ..,d_n]$ of this initial population is generated by means of a `random_labeling` procedure as shown in figure 3 and described below:

- Variable (V_i) selection: next variable to be labeled is randomly selected.
- Value (domain d_i) selection: from the domain of the variable V_i, which ranges from *Min* to *Max*, two values *Low* and *Up* are randomly chosen (*Min* <= *Low* <= *Up* <= *Max*). Then V_i is constrained to a domain d_i, which is a reduction of the original domain. This reduction may be performed in two different ways: one reduces the domain removing values from the top and/or bottom, and the other tries to reduce the domain towards one of its boundaries. For each variable we randomly select one of this two ways, using a random real number between 0.0 and 1.0, and a parameter, called *boundary_prob*, that specifies the probability that the "boundary" domain reduction is chosen.

  If the first way of domain reduction is chosen, the new domain d_i is chosen randomly, in a non deterministic way, from the following domains:
  - *Low..Up*
  - *Low..Max*
  - *Min..Up*

  If boundary reduction is chosen, the new domain d_i is chosen randomly, in a non deterministic way, from the following domains:
  - *Min..Low*
  - *Up..Max*

```
function random_labeling (Vars : list of fd_vars) : chromosome;
begin
  randomly select a variable V_i from Vars
  fdvar_range(V_i,Min,Max);
  choose_randomly(Low,Up) from [Min to Max];
  if  random_number < border_prob  then  begin
    try secuencialy in any order:
        V_i :: Low..Up
        V_i :: Low..Max
        V_i :: Min..Up
  end
  else begin
    try secuencialy in any order:
        V_i :: Min..Low
        V_i :: Up..Max
  end
  if  all fail  then  V_i :: Min..Max
  fdvar_domain(V_i,d_i)
  return d_i ∪ random_labeling(Vars / V_i);
end.
```

Figure 3.

2.4 Evaluation of the population.

The population is evaluated every generation, computing a fitness value for each chromosome. Fitness indicates how good a chromosome is as a potential solution to the problem, so the domain associated to the cost expression to optimize is the basis for the computation of this fitness value. The probability of survival and reproduction of a chromosome is directly proportional to its fitness value.

Figure 4 shows the main steps of the computation of the fitness value. The fitness of each chromosome is computed as $f(d_C,[d_1, ..,d_n])$, a function of d_C, the domain of the cost function for that particular solution, and the list of d_i's, the remaining domains of the variables. The values L_{d_C} and U_{d_C}, the lower and upper bounds of d_C respectively, are the main contribution to the fitness function (`basic_fitness`), but there is also two penalty components, one (`penalty_cost`) depending on Sd_C, the size of the domain d_C, and other (`penalty_vars`) depending on the sum of the sizes of d_i's. The introduction of these penalty factors favors those chromosomes closer to be ground, so that the algorithm tends to converge to a ground solutions.

Parameters *penal_cost* and *penal_vars* may take any real value from 0.0 to 1.0. They weights the penalty introduced to those chromosomes not completely ground. Then, as shown in Figure 2, the `total_fitness` value is computed as the sum of all fitness values, and finally, we compute the relative fitness (X_i`.fit / total_fitness`) for each chromosome, and its accumulated relative fitness value, used for the random selection of a chromosome with a probability proportional to its fitness.

```
function fitness(Cost: function; [d1,...,dn]: list of domains) : real;
begin
   cost_range_size(Cost,Ldc,Udc,Sdc);
   Fit1:= basic_fitness(Ldc,Udc);
   Pc := penalty_cost(Sdc);          /* 0.0 to 1.0 */
   Pv := penalty_vars(Σ size(di));   /* 0.0 to 1.0 */
   Fit2 := (1-Pc*penal_cost) * Fit1;
   Fit3 := (1-Pv*penal_vars) * Fit2;
   return Fit3
end.
```

Figure 4.

2.5 Selection of chromosomes to survive.

Some chromosomes from population *P(t-1)* will be present in population *P(t)*. This set of chromosomes is randomly chosen, but as we want the population to converge to a good solution, chromosomes with a higher fitness value are more likely to be chosen. Selecting in a random fashion allows some "no good" chromosomes to be selected for survival and crossover. This is an important point of the evolutionary mechanism: bad solutions cannot be simply dropped because they may eventually lead to a good solution. Figure 5 shows the algorithm that randomly chooses a chromosome of the population with a probability proportional to its relative fitness. This algorithm is also used to select parent chromosomes for the crossover operator.


```
function select_chromosome (P : population) : chromosome;
begin
   R := random number between 0.0 and 1.0.
   i := 1;
   while  R > Xi.arf  do  i:=i+1;
   return Xi
end.
```

Figure 5.

2.6 Genetic operators.

Genetic operators generate the new chromosomes that will be added to population *P(t)* from chromosomes from population *P(t-1)*. The design of these operators is a crucial point, as they must guarantee that new individuals "inherits" the good properties of their parents, and also must allow the exploration of new areas of the search space.

In simple evolutionary programs, like classical genetic algorithms, chromosomes are coded as bit strings. Binary mutation just inverts some randomly selected bits, and binary crossover concatenates two substrings obtained from splitting the parents. However, in evolutionary programs chromosomes are complex data structures, and genetic operators are much elaborated. The operators used in our system are quite more complex than the classical ones, not just because they work over a complex data structure (a list of finite integer domains), but mainly because they trigger the local consistency and constraint propagation techniques embedded in CLP(*FD*). Anyway, because of intuitive similarities, we cluster the operators in the standard two classes, mutation and crossover. We have included two mutation operators, which create new solutions by a small change in a single chromosome, and a crossover operator, which create a new solution by combining two chromosomes.

2.6.1 Crossover.

"Dead" chromosomes (those not selected for survival) are replaced by new chromosomes generated by means of the crossover operator. Dead chromosomes are not actually replaced until all new chromosomes are generated, so dead chromosomes may also be selected to generate a new ones by crossover. As shown in Figure 2, parent chromosomes are chosen using the `select_chromosome` procedure (described in subsection 2.5), which randomly selects a chromosome with a probability proportional to its fitness.

Figure 6 shows the algorithm used for `crossover`. Given two chromosomes the crossover operator generates a new solution which is an approximate mixture of the two parents. From two chosen parents, $X_i = [d_1^i, ..,d_n^i]$ and $X_j = [d_1^j, ..,d_n^j]$, a new chromosome $X = [d_1, ..,d_n]$ is generated by means of a `crossover` labeling procedure as follows: (Keep in mind that whenever a domain variable is forced to modify its domain, local consistency and constraint propagation is triggered)

1. Variable (V_k) selection: next variable to be labeled is randomly selected, being D_k its associated domain.
2. Value (domain) selection: Value (domain d_k) selection: if $d_k^i \cap d_k^j \neq \emptyset$ then d_k is randomly assigned the domain intersection or the domain union between d_k^i and d_k^j, with a probability *xov_inter_prob* in favor of the intersection. If $d_k^i \cap d_k^j = \emptyset$ then we try to assign to d_k, in random order, d_k^i or d_k^i. If all trials fail, d_k is assigned D_k

```
function crossover(Vars:list of fd_vars;X_i,X_j :chromosome):chromosome;
begin
  randomly select a variable V_k from Vars;
  fdvar_domain(V_k,D_k)
  d_k^i := k_th(X_i)
  d_k^j := k_th(X_j)
  if  d_k^i ∩ d_k^j ≠ ∅  then begin
    R := random number between 0.0 and 1.0
    if  R > xov_inter_prob   then d_k := d_k^i ∩ d_k^j
                             else d_k := d_k^i ∪ d_k^j
  end
  else begin
    try secuencialy in random order:
      d_k := d_k^i
      d_k := d_k^j
  end
  if  all fail  then  d_k := D_k;
  V_k :: d_k                  /* may fail and backtrack */
  return  d_k U crossover(Vars / V_k, X_j / d_k^i, X_j / d_k^j)
end;
```

Figure 6.

2.6.2 Mutation.

Mutation is the unary genetic operator that transforms a single chromosome in a new chromosome. It plays the role of "jumping" to unexplored areas of the search space. We have included two mutation operators, which may be applied to any chromosome in the population with a probability of *mut_prob1* and *mut_prob2*, respectively. Operator 1 intends to expand the domain of the chosen variable, whereas operator 2 intends to "move" the domain of a variable to new values. Figure 7 shows the algorithms for the mutation operators.

1. Operator 1:

   Given a chromosome $X = [d_1, .., d_n]$, we generate a new mutated chromosome $X' = [d_1,..., d_k',.., d_n]$, assigning to each variable V_i the domain d_i, except a randomly chosen variable V_k, which is constrained to a new domain d_k', computed from its associated domain D_k, ranging from *Min* to *Max*, and d_k, ranging from *Low* to *Up* (*Min* <= *Low* <= *Up* <= *Max*). d_k' is chosen, randomly, in a non deterministic way, from:

   - *Up..Max*
   - *Min..Low*

   If both trials fail, the domain of V_k is left unchanged.

2. Operator 2:

   Given a chromosome $X = [d_1, .., d_n]$, we generate a mutated new chromosome $X' = [d_1,..., d_k',.., d_n]$, assigning to each variable V_i the domain d_i, except a randomly chosen variable V_k, which is constrained to a new domain d_k', computed from its associated domain D_k, ranging from *Min* to *Max*, and d_k, ranging from *Low* to *Up* (*Min* <= *Low* <= *Up* <= *Max*). d_k' is chosen, randomly, in a non deterministic way, from:

   - *Min..Up*
   - *Low..Max*

```
function mutation(Vars : list of fd_vars; X : chromosome) :chromosome;
begin
  R := random number between 0.0 and 1.0
  if  R > mut_prob1  then X' := mutation1(Vars,X)
                     else X' := X
  R := random number between 0.0 and 1.0
  if  R > mut_prob2  then X'' := mutation2(Vars,X')
                     else X'' := X'
  return  X''
end;

function mutation1(Vars : list of fd_vars; X : chromosome):chromosome;
begin
  randomly select a variable V_k from Vars
  for  i:=1 to length(Vars)  do
    if  i <> k  then  V_i :: d_i
  fdvar_domain(V_k,d_k')
  return  (X with d_k replaced by d_k')
end;

function mutation2(Vars : list of fd_vars; X : chromosome):chromosome;
begin
  randomly select a variable V_k from Vars
  for  i:=1 to length(Vars)  do
    if  i <> k  then  V_i :: d_i
  d_k := k_th(X);
  fdvar_range(V_k,Min,Max)
  domain_range(d_k,Low,Up);
  try in any order:
    d_k' := Min..Low
    d_k' := Up..Max
  V_k :: d_k'
  if both fail then  d_k' := Min..Max
  return  (X with d_k replaced by d_k')
end;
```

Figure 7.

2.7 Termination condition.

The termination condition is the disjunction of the following factors:

- reaching the maximum number of iterations specified by the parameter *max_iter*.
- reaching a user specified time-out.
- obtaining a chromosome with a user specified cost.
- reaching a hopelessly invariant population.

2.8 Extracting the best solution.

Once the termination condition is met, we must extract the best chromosome from the population. Some or even all chromosomes in the population may not be completely ground solutions, so we must use a heuristic to extract the best ground solution present in the population. Figure 8 shows the algorithm used to perform this search in the population.

First we look in the population for the best "ground" chromosome, that is, the best fitted chromosome $X_{g1} = [d_1, ..,d_n]$ such that for all i, d_i is a singleton domain, and also for the chromosome X_{ng} that, not being completely ground, offers a higher fitness. From chromosome X_{ng}, we generate a ground solution X_{g2} by means of a classical labeling procedure, and a pool of chromosomes by means of a random labeling procedure. From this pool, we extract the best ground solution X_{g3} and the best non ground chromosome. The latter gives place to a ground solution X_{g4} by means of a classical labeling procedure. The solution offered as the final near-optimal labeling for the input variables will be the one which offers a better value for the objective function to optimize among the four ground solutions X_{g1}, X_{g2}, X_{g3} and X_{g4}.

```
function best_solution (Vars : list of fd_variables;
                        P : population ) : list of integers;
begin
   Xg1 := best fitted ground solution in P
   Xng := best fitted non completely ground chromosome in P
   Xg2 := labeling(Xng)
   Vars' := Vars updated with Xng
   P' := generated by random_labeling(Vars');
   Xg3 := best fitted ground solution in P'
   X'ng := best fitted non completely ground chromosome in P'
   Xg4 := labeling(X'ng);
   return max_fitness(Xg1,Xg2,Xg3,Xg4)
end;
```

Figure 8.

2.9 Parameters.

As seen throughout this section, an evolutionary algorithm uses some global parameters indicating the population size (*pop_size*), proportion of chromosomes to survive from one generation to the next one (*prop_surv*), the maximum numbers of generations to run (*max_iter*), two penalty percentages (*penal_cost* and *penal_vars*), and four different probabilities that tune the behavior of the genetic operators (*boundary_prob*, *xov_inter_prob*, *mut_prob1*, *mut_prob2*). The values of these parameters affects dramatically the performance of the evolutionary algorithm, and there are no general values that performs optimally for every benchmark. Table 1 shows the usual range for each of the parameters.

pop_size	10 - 100	*max_iter*	50 - 1000
boundary_prob	0.20 - 0.70	*penal_cost*	0.05 - 0.40
penal_vars	0.20 - 0.90	*prop_surv*	0.30 - 0.80
xov_inter_prob	0.30 - 0.70	*mut_prob1*	0.01 - 0.15
mut_prob2	0.01 - 0.15		

Table 1.

Parameters should be initialized every time the evolutionary algorithm is invoked in accordance to the initial domains of the variables to be labeled and the function cost. Also, parameters tuning the behavior of the genetic operators should be tuned every fixed number of iterations depending on the evolution of the population. It is still under development a heuristic-guided self adaptive parameter tuning feature, which is essential to achieve our goal, a self contained optimization predicate for constraint logic programming over finite integer domains.

3 Example.

A prototype of the system has been implemented in Prolog using the logic programming environment ECLiPSe, which offers several facilities for the integration of extensions in logic programming [Ecli94]. The constraint handling itself is provided as an extension by means of a library. We pretend that the final version of our system will also be deliverable as a library for the ECLiPSe system.

In this section we describe a simple example of the system solving a transportation problem. It seeks the determination of a minimum cost transportation plan for a single commodity from n sources to k destinations. The amount of supply at source i is *sour(i)*, and the demand at destination j is *dest(j)*. The unit transportation cost between source i and destination j is *cost(i,j)*. The amount transported from source i to destination j is X_{ij}. The constraints and the objective function are:

$\sum^{k}_{j=1} X_{ij} = sour(i)$ for i=1, ..., n

$\sum^{n}_{i=1} X_{ij} = dest(j)$ for j=1, ..., k

$f = \sum X_{ij} * cost(i,j)$

Test data correspond to a n=7, k=7 problem taken from [Mic94]. Figure 9 shows the evolution of the cost of best chromosome in the population vs. the number of generations.

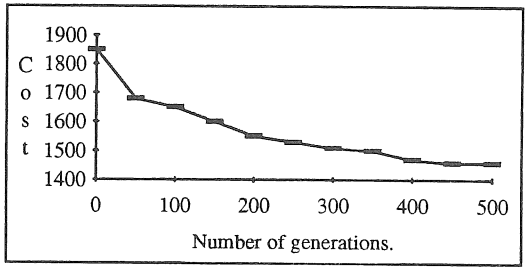

Figure 8.

The shape of the curve is characteristic of evolution programs. Chromosomes in the randomly generated initial population have very poor fitness values, but in a few iterations good solutions are generated. Then, new better solutions take more generations to appear, and also all chromosomes in the population slowly tend to converge to the same near-optimal solution.

We are currently working on a set of benchmarks of combinatorial optimization problems: job-shop scheduling, traveling salesman, graph partitioning, assembly line sequencing, and time tabling. First results lead us to expect that a final version of the system will be competitive with classical optimization strategies like branch and bound.

4 Conclusions and future work.

We have introduced a model to integrate evolutionary algorithms in constraint logic programming over finite integer domains in order to perform the optimal labeling phase of combinatorial search optimization problems. Chromosomes represent not completely ground solutions to the problem to guarantee a wider covering of the search space. A set

of genetic operators have been designed in accordance to the particular characteristics of constraint logic programming. We have developed a prototype using the facilities provided by the logic programming environment ECLiPSe and performed some experiments that allow us to expect that a final version of the system will be competitive with other optimization methods, like branch and bound, when applied to problems with a vast search space.

This work is still in its first stage. We believe that the integration of the evolutionary programming paradigm in constraint logic programming for optimization purposes has a promising future. The implemented prototype has many drawbacks to be fixed, more investigation is to be done to design better genetic operators and a self adaptive parameter tuning is still missing. Besides this problems, future work will emphasize in exploiting the great possibilities of parallelism that the integration evolutionary algorithms and CLP offers.

References

[Din90] Dincbas, M., Simonis H., and Van Hentenryck. "*Solving Large Combinatorial Problems in Logic Programming*". Journal of Logic Programming 8. 1990

[Gol89] Goldberg, D.E., "*Genetic algorithms in search, Optimization and Machine Learning*". Addison-Wesley, Reading, MA, 1989.

[Hol75] Holland, J.H., "*Adaptation in Natural and Artificial Systems*". University of Michigan Press, Ann Arbor, 1975.

[Mic93] Michalewicz, Z., and Attia, N., "*Evolutionary optimization of Constrained Problems*". Proc. of the 3rd Ann. Conf. on Evolutionary Programming, La Jolla, CA, 1993.

[Mic94] Michalewicz, Z., "*Genetic algorithms + Data Structures = Evolution Programs*". Second Edition, Springer-Verlag.1994.

[Min92] Minton, S., Johnston M.D., Philips A.B., and Laird, P., "*Minimizing conflicts: a Heuristic Repair Method for constraint satisfaction and scheduling problems*". Artificial Intelligence 58. 1992.

[VH89] Van Hentenryck, P. "*Constraint Satisfaction in Constraint Logic Programming*". The MIT Press. 1989.

[VH92] Van Hentenryck, P., Deville, Y., and Teng C.M., "*A generic Arc-consistency Algorithm and its Specializations*". Artificial Intelligence 57. 1992.

[Wal94] Wallace M., "*Constraints in Planing, Scheduling and Placement Problems*", In Constraint Programming,. Springer-Verlag, 1994.

[Ecli94] "*ECLiPSe Extensions User Manual*". ECRC GmbH. July 1994.

Constraint Systems for Pattern Analysis of Constraint Logic-Based Languages

Roberto Bagnara
Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa
bagnara@di.unipi.it
Phone: 050/887267 Fax: 050/887226

Abstract

Pattern analysis consists in determining the shape of the set of solutions of the constraint store at some program points. Our basic claim is that pattern analyses can all be described within a unified framework of constraint domains. We show the basic blocks of such a framework as well as construction techniques which induce a hierarchy of domains. In particular, we propose a general methodology for domain combination with asynchronous interaction. The interaction among domains is asynchronous in that it can occur at any time: before, during, and after the product operation in a completely homogeneous way. That is achieved by regarding semantic domains as particular kinds of (ask-and-tell) constraint systems. These constraint systems allow to express communication among domains in a very simple way. The techniques we propose allow for smooth integration within an appropriate framework for the definition of non-standard semantics of constraint logic-based languages. The effectiveness of this methodology is being demonstrated by a prototype implementation of CHINA, a CLP($\mathcal{H}$, N) analyzer we have developed.

Keywords: Constraint Systems, Constraint-based Languages, Data-flow Analysis, Abstract Interpretation.

1 Introduction

Pattern analysis for constraint logic-based languages consists in determining the shape of the set of solutions of the constraint store at some program point. For usual applications (most prominently, program specialization) the interesting program points are procedure calls and procedure (successful) exits.

In the case of Prolog, pattern analysis has been extensively studied (see [9] for a summary of this work). In the case of CLP, besides the generalization to CLP($\mathcal{H}$) of the ideas and techniques used for Prolog, not much has been done. A key observation here is that the shape of solutions can be conveniently described by constraints. Thus the CLP framework is general enough to encompass (some of) its own data-flow analyses. Intuitively, this is done by replacing the standard constraint domain with one suitable for expressing the desired information. This fundamental aspect was brought to light in [5] and elaborated in [12].

For languages of the kind of CLP(N), where N is some numerical domains, the first steps towards pattern analysis were moved in [3, 4]. [2] describes some of the more important applications of such analyses. The work done in this field is being generalized to CLP($\mathcal{H}$, N) languages, integrating numerical and symbolic pattern analysis. This is done with a variety of techniques,

including depth-k abstraction. A more restricted kind of integration has recently been described in [17]. Here, the numerical part is essentially the one proposed in [3].

Now, instead of directly describing the techniques employed in [3, 4, 2, 17], we concentrate on what is missing from them: a general notion of constraint domain which allows one to adequately describe both the "logical part" of concrete computations (e.g. answer constraints) and as much pattern analyses (e.g. the shape of those answer constraints) we can think about.

We believe that it is possible to describe every pattern analysis within a unified framework of constraint domains. In particular we wish the framework being able to accommodate approximate inference techniques whose importance relies on very practical considerations, such as representing good compromises between precision and computational efficiency. Some of these techniques will be sketched in the sequel.

Then, what will be needed is a generalized algebraic semantics for constraint logic programs, parameterized with respect to an underlying constraint domain. The main advantages of this approach [12] are that: (1) different instances of CLP can be used to define non-standard semantics for constraint logic programs; and (2) the abstract interpretation of CLP programs can be thus formalized inside the CLP paradigm.

Let us concentrate on constraint domains for pattern analysis. They are algebraic structures of the kind

$$\bar{\mathcal{D}} = \langle \mathcal{D}, \preceq, \otimes, \oplus, \{\exists_\Delta\}, \mathbf{0}, \mathbf{1}, \{d_{\bar{X}\bar{Y}}\}\rangle, \tag{1}$$

where[1] $\mathcal{D}$ is the set of constraints expressing the properties of interest. $\mathcal{D}$ is partially ordered with respect to $\preceq$ which, intuitively, relates the information content of constraints: $C_1 \preceq C_2$ means that "C_1 is more precise than C_2". $\otimes$ and $\oplus$ are binary operators modeling conjunction and (weak) disjunction. $\{\exists_\Delta\}$ is a family of unary operators, indexed over finite subsets of variables, modeling projection of constraints onto designated sets of variables. $\mathbf{0}$ and $\mathbf{1}$ represent, intuitively, the class of unsatisfiable constraints and the class of non-constraints (i.e. those which do not provide any information), respectively. The family of distinguished elements $\{d_{\bar{X}\bar{Y}}\}$, indexed on pairs of n-tuple of variables, allows to model parameter passing.

In this setting, data-flow analysis is then performed (or at least justified) through abstract interpretation [8, 9], i.e., "mimicking" the program run-time behavior by "executing" it, in a finite way, on an approximated (abstract) constraint domain. We will thus have two constraint domains of the form (1): the "concrete" and the "abstract" one. Following a generalized semantic approach, the concrete and abstract semantics are more easily related, being instances (over two different constraint systems) of the same generalized semantics, which is entirely parametric on a constraint domain. Thus, to ensure correctness, it will be sufficient to exhibit an "abstraction function" α which is a semimorphism between the constraint domains [10].

In this paper we describe a hierarchy of constraint systems which capture all the pattern analyses we know of, as well as the "concrete" collecting semantics they abstract. The basis is constituted by a set of finite constraints, each expressing some partial information about a program execution's state. Once this is given (simple constraint systems, Section 2), we provide standard ways of representing and composing finite constraints (determinate constraint systems, Section 3). Then we can have the notion of *dependency* built into the constraint system (ask-and-tell constraint systems Section 4). Another construction is the one which allows us to treat *disjunction* (powerset constraint systems, Section 5). Finally, in Section 6 we sketch how to achieve combination of domains by considering dependencies within *product constraint systems*. We feel that, indeed, this is one of more important contributions of this paper.

For the sake of simplicity we will present constraint systems omitting the distinguished elements modeling parameter passing. For most applications $d_{\bar{X}\bar{Y}}$ is simply a constraint expressing some sort of equivalence between X and Y. We disregard them also because, differently from [12], we do not require them to satisfy any interesting algebraic property.

[1] For space reasons we omit many details.

2 Simple constraint systems

The basic blocks of our construction are *simple constraint systems* (or s.c.s.), very similar to those of [19], but with a *totally uninformative* token ($\top$) as in [20].

Definition 2.1 (Simple constraint system.) *A* simple constraint system *is a structure* $\langle \mathcal{C}, \vdash, \bot, \top\rangle$, *where* $\mathcal{C}$ *is a set of (not better specified) constraints,* $\bot \in \mathcal{C}$, $\top \in \mathcal{C}$, *and* $\vdash \subseteq \wp_f(\mathcal{C}) \times \mathcal{C}$ *is a compact* entailment relation *such that, for each* $C, C' \in \wp_f(\mathcal{C})$ *and* $c, c' \in \mathcal{C}$:

$$E_1.\ \ c \in C \Rightarrow C \vdash c, \qquad E_3.\ \ (C \vdash c) \wedge (\forall c' \in C : C' \vdash c') \Rightarrow C' \vdash c,$$
$$E_2.\ \ C \vdash \top, \qquad E_4.\ \ \{\bot\} \vdash c$$

We consider also the extension $\vdash \subseteq \wp(\mathcal{C}) \times \wp(\mathcal{C})$ *such that, for each* $C, C' \in \wp(\mathcal{C})$,

$$C \vdash C' \quad \Leftrightarrow \quad \forall c' \in C' : \exists C'' \subseteq_f C \,.\, C'' \vdash c'.$$

It is clear that condition E_1 implies reflexivity of $\vdash$, while condition E_3 amounts to transitivity. E_2 qualifies $\top$ as the least informative token: it will be needed just as a "marker" when the *product* of simple constraint systems will be considered (see Section 6). E_4 ensures that $\mathcal{C}$ is a finitely generable element.

In general, describing the "standard" semantics of a CLP(X) language is an easy matter. Let T be the theory which corresponds to the domain X [15]. Let D be an appropriate set of formulas in the vocabulary of T closed under conjunction and existential quantification. Define $\Gamma \vdash c$ iff Γ entails c in the logic, with non-logical axioms T. Then $(D, \vdash)$ is the required simple constraint system. For CLP($\mathcal{H}$) (i.e. pure Prolog) one takes the Clark's theory of equality. For CLP($\mathbb{R}$) the theory RCF of real closed fields will do the job.

However, describing "standard" constraint domains is not the reason which motivated our work. Here are the original motivations.

2.1 Pattern analysis for numeric domains

The analysis described in [3, 4, 2] is based on constraint inference (a variant of constraint propagation) [11]. This technique, developed in the field of artificial intelligence, has been applied to temporal and spatial reasoning [1, 21].

Let us focus our attention to arithmetic domains, where the constraints are binary relations over expressions. Let E be the set of arithmetic expressions of interest and I the set of intervals over some computable set of numbers (e.g. rational or floating point numbers). Then our constraints are given by

$$\mathcal{C} = \{\, e_1 \bowtie e_2 \mid \bowtie \in \{=, \neq, \leq, <, \geq, >\}, e_1, e_2 \in E \,\} \cup \{\, e \lhd I \mid e \in E, I \in \mathsf{I} \,\}.$$

The meaning of the constraint $e \lhd I$ is the obvious one: any value the expression e can take is contained in I. Thus $\mathcal{C}$ provides a mixture of qualitative (relationships) and quantitative (bounds) knowledge.

Now, the approximate inference techniques we are interested in can be encoded into a consequence relation $\vdash$ over $\mathcal{C}$. Let us see some of them. The most trivial one is *symmetric closure*: $\{e_1 \bowtie e_2\} \vdash e_2 \bowtie^{-1} e_1$, where $\bowtie^{-1}$ is the inverse of $\bowtie$ (e.g., $<$ is the inverse of $>$, $\geq$ of $\leq$ and so on). A more interesting qualitative technique is *transitive closure*, allowing inferences like $A < C$ from $A \leq B$ and $B < C$. It is formalized by axioms of the form $\{e_1 \leq e_2, e_2 < e_3\} \vdash e_1 < e_3$. A classical quantitative technique is *interval arithmetic* which allows to infer the variation interval of an expression from the intervals of its sub-expressions. Let $f(e_1, \ldots, e_k)$ be any arithmetic expression having $e_1, \ldots, e_k$ as subexpressions. Then $\{f(e_1, \ldots, e_k) \lhd I, e_1 \lhd I_1, \ldots, e_k \lhd I_k\} \vdash f(e_1, \ldots, e_k) \lhd \ddot{f}(I_1, \ldots, I_k)$, where $\ddot{f}: \mathsf{I}^k \to \mathsf{I}$ is such that for each $x_1 \in I_1, \ldots, x_k \in I_k$, $f(x_1, \ldots, x_k) \in \ddot{f}(I_1, \ldots, I_k)$. An example inference is: $A \lhd [3, 6) \wedge B \lhd [-1, 5] \vdash A + B \lhd [2, 11)$.

Another technique is *numeric constraint propagation*, which consists in determining the relationship between two expressions when their associated intervals do not overlap, except possibly at their endpoints. The associated family of axioms is $\{e_1 \lhd I_1, e_2 \lhd I_2\} \vdash e_1 \bowtie e_2$, with the side condition $\forall x_1 \in I_1, x_2 \in I_2 : x_1 \bowtie x_2$. For example, if $A \in (-\infty, 2]$, $B \in [2, +\infty)$, and $C \in [5, 10]$, we can infer that $A \leq B$ and $A < C$. It is also possible to go the other way around, i.e., knowing that $U < V$ may allow to refine the intervals associated to U and V so that they do not overlap. We call this *weak interval refinement*: $\{e_1 \bowtie e_2, e_1 \lhd I_1, e_2 \lhd I_2\} \vdash e_1 \lhd I_1'$, where I_1' is obtained by shrinking I_1 so to ensure that $x_1 \in I_1'$ iff $x_1 \in I_1 \wedge \exists x_2 \in I_2 \,.\, x_1 \bowtie x_2$.

In summary, by considering the transitive closure of $\vdash$ and with some minor technical additions we end up with a simple constraint system which characterizes precisely the combination of the above (and possibly other) techniques.

3 Determinate constraint systems

By axioms E_1 and E_3 of Definition 2.1 the entailment relation of a simple constraint system is a preorder. Now, instead of considering the quotient poset with respect to the induced equivalence relation, a particular choice of the equivalence classes' representatives is made: closed sets with respect to entailment. This representation is a very convenient domain-independent strong normal form for constraints.

Definition 3.1 (Elements.) [19] *The* elements *of an s.c.s.* $\langle \mathcal{C}, \vdash, \bot, \top \rangle$ *are the* entailment-closed *subsets of* $\mathcal{C}$*, that is, those* $C \subseteq \mathcal{C}$ *such that* $\exists C' \subseteq_f C \,.\, C' \vdash c$ *implies* $c \in C$*. The set of elements of* $\langle \mathcal{C}, \vdash \rangle$ *is denoted by* $|\mathcal{C}|$.

The poset of elements is thus given by $\langle |\mathcal{C}|, \supseteq \rangle$. Notice that we deviate from [19] in that we order our constraint systems in the dual way.

Definition 3.2 (Inference map, finite elements.) *Given a simple constraint system* $\langle \mathcal{C}, \vdash, \bot, \top \rangle$*, the* inference map *of* $\langle \mathcal{C}, \vdash, \bot, \top \rangle$ *is the function* $\rho : \wp(\mathcal{C}) \to \wp(\mathcal{C})$ *given, for each* $C \subseteq \mathcal{C}$*, by* $\rho(C) = \{ c \mid \exists C' \subseteq_f C \,.\, C' \vdash c \}$*. It is well known that* ρ *is a kernel operator, over the complete lattice* $\langle \wp(\mathcal{C}), \supseteq \rangle$*, whose image is* $|\mathcal{C}|$*. The image of the restriction of* ρ *onto* $\wp_f(\mathcal{C})$ *is denoted by* $|\mathcal{C}|_0$*. Elements of* $|\mathcal{C}|_0$ *are called* finitely generated constraints *or simply* finite constraints.

From here on we will only work with finitely generated constraints, since we are not concerned with infinite behavior of CLP programs. The next step in our construction is about *determinate constraint systems* (or d.c.s.).

Definition 3.3 (Determinate constraint system.) *Let* $\mathcal{S} = \langle \mathcal{C}, \vdash, \bot, \top \rangle$ *be a simple constraint system. Let* $\mathbf{0}, \mathbf{1} \in |\mathcal{C}|_0$, $\otimes : |\mathcal{C}|_0 \times |\mathcal{C}|_0 \to |\mathcal{C}|_0$*, and* $\vdash \subseteq |\mathcal{C}|_0 \times |\mathcal{C}|_0$ *be given, for each* $C_1, C_2 \in |\mathcal{C}|_0$*, by*

$$\begin{array}{lcl@{\qquad}lcl} \mathbf{0} & = & \mathcal{C}, & C_1 \otimes C_2 & = & \rho(C_1 \cup C_2), \\ \mathbf{1} & = & \rho(\emptyset), & C_1 \vdash C_2 & \Leftrightarrow & C_1 \otimes C_2 = C_1. \end{array}$$

The projection operators $\exists_\Delta : |\mathcal{C}|_0 \to |\mathcal{C}|_0$ *are given, for each* $\Delta \subseteq_f Vars$ *and each* $C \in \wp(\mathcal{C})$*, by*

$$\exists_\Delta C = \rho(\{ c \in C \mid FV(c) \subseteq \Delta \}).$$

Finally, let $\oplus : |\mathcal{C}|_0 \times |\mathcal{C}|_0 \to |\mathcal{C}|_0$ *be an operator enjoying the following properties:*

J_1. $\langle |\mathcal{C}|_0, \oplus, \mathbf{0} \rangle$ *is a commutative and idempotent monoid;*

J_2. *for each* $C_1, C_2 \in |\mathcal{C}|_0$, $C_1 \vdash C_1 \oplus C_2$ *and* $C_2 \vdash C_1 \oplus C_2$.

We will refer to the structure $\langle |\mathcal{C}|_0, \vdash, \otimes, \oplus, \{\exists_\Delta\}, \mathbf{0}, \mathbf{1} \rangle$ *as the* determinate constraint system *over* $\mathcal{S}$ *and* $\oplus$*. The relation* $\sqsubseteq$ *induced by* $\oplus$ *over* $|\mathcal{C}|_0$ *is given, for each* $C_1, C_2 \in |\mathcal{C}|_0$*, by* $C_1 \sqsubseteq C_2$ *iff* $C_1 \oplus C_2 = C_2$*. The relations* $\vdash$ *and* $\sqsubseteq$ *are referred to, respectively, as the* approximation ordering *and the* computational ordering *of the determinate constraint system.*

Observe that the required conditions on $\oplus$ are quite reasonable. The purpose of $\oplus$ is that of "merging" the information originating from different paths in the semantic construction. In this view, axiom J_1 is very natural: associativity and commutativity amount to say that we can merge paths in any order, idempotence means that we do not loose precision blindly, and $\mathbf{0}$ being the monoid unit accounts for the ability of discarding failed computation paths. Condition J_2 states the correctness of the merge operation, characterizing it as a (not necessarily least) upper bound operator with respect to the approximation ordering.

Notice that the distinction between approximation ordering and computational ordering is important. We assume that our semantics are defined as (approximations of) least fixpoints of some operator[2] ϕ. So, while the approximation ordering, in general abstract interpretation, specifies the relative precision of program properties (e.g. entailment of constraints in our particular case), the computational ordering holds among the iterates $\phi^k(\bot)$ during the fixpoint computation. The case where the two orderings coincide (e.g. in [12]) is thus to be considered a special one. In our treatment, keeping them distinct allows for more freedom in the choice of the merge operator.

Since the set of finite computation paths is, in general, denumerably infinite, we consider also the following strengthening of Definition 3.3.

Definition 3.4 (Closed d.c.s.) *A d.c.s.* $\langle |\mathcal{C}|_0, \vdash, \otimes, \oplus, \{\exists_\Delta\}, \mathbf{0}, \mathbf{1} \rangle$ *is said* closed *iff it satisfies*

J_3. *for each family* $\{C_i \in |\mathcal{C}|_0\}_{i \in \mathbb{N}}$, $\bigoplus_{i \in \mathbb{N}} C_i = C_1 \oplus C_2 \oplus \cdots$ *exists and is unique in* $|\mathcal{C}|_0$*; moreover, associativity, commutativity, and idempotence of* $\oplus$ *apply to denumerable as well as to finite families of operands.*

So, the operation of merging together the information coming from all the computation paths always makes sense in a closed determinate constraint system. Notice however that property J_3 is only necessary when the semantic construction requires it. This will never happen when considering "abstract" semantic constructions formalizing data-flow analyses (which are finite in nature). In these cases the idea of merging infinitely many pieces of information is a nonsense in itself.

Determinate constraint systems enjoy several properties. Here are some elementary ones: $\sqsubseteq$ is a partial order and $C_1 \sqsubseteq C_2$ implies $C_1 \vdash C_2$; $\otimes$ and $\oplus$ are componentwise monotone with respect to $\vdash$ and $\sqsubseteq$, respectively; $\mathbf{0}$ is an annihilator for $\otimes$, while $\mathbf{1}$ is a unit for $\otimes$ and an annihilator for $\oplus$. Finally, for absorption laws we have $C_2 = (C_1 \oplus C_2) \otimes C_2$ and $C_2 \vdash (C_1 \otimes C_2) \oplus C_2$. At a higher level, here is the situation.

Theorem 3.1 *Let* $\mathcal{D} = \langle |\mathcal{C}|_0, \vdash, \otimes, \oplus, \{\exists_\Delta\}, \mathbf{0}, \mathbf{1} \rangle$ *be a determinate constraint system. Then the structure* $\langle |\mathcal{C}|_0; \vdash, \mathbf{0}, \mathbf{1}, \otimes \rangle$ *is a bounded meet-semilattice and* $\langle |\mathcal{C}|_0; \sqsubseteq, \mathbf{0}, \mathbf{1}, \oplus \rangle$ *is a join-semilattice. Moreover, if* $\mathcal{D}$ *is complete, then* $\langle |\mathcal{C}|_0; \sqsubseteq, \mathbf{0}, \mathbf{1}, \oplus \rangle$ *is a (join-) complete lattice.*

Notice that $\langle |\mathcal{C}|_0, \otimes, \oplus, \mathbf{0}, \mathbf{1} \rangle$, in general, is not a lattice. Both $\otimes$ and $\oplus$ are associative, commutative, and idempotent. But, as stated above, while one of the absorption laws holds, only one direction of the dual law is generally valid. In particular $\otimes$ is not required to be componentwise monotone with respect to $\sqsubseteq$, and $\oplus$ might be not componentwise monotone with respect to $\vdash$. Observe also that $\oplus$ does not distribute, in general, over $\otimes$, as this would imply the equivalence of the two absorption laws.

[2] For example, if we choose a bottom-up (backward) semantic construction for CLP, this will be an immediate consequence operator T_P parameterized on the underlying constraint system [12]. We disregard this issues here, as we concentrate on the construction of constraint domains.

4 Ask-and-tell constraint systems

We now consider constraint systems having additional structure. This additional structure allows to express, at the constraint system level, that the imposition of certain constraints must be delayed until some other constraints are imposed. In [18] similar constructions are called *ask-and-tell constraint systems*. In our construction, ask-and-tell constraint systems are built from determinate constraint systems by regarding some kernel operators as constraints. We follow [18] in considering cc as *the* language framework for expressing and computing with kernel operators. For this reason we will present kernel operators as cc agents. For our current purposes we only need a very simple fragment of the determinate cc language: the one of *finite* cc *agents*. This fragment is described in [19] by means of a declarative semantics. Here we give an operational characterization which is better suited to our needs.

Definition 4.1 (Finite cc agents: syntax.) *A* finite cc agent *over a simple constraint system* $\mathcal{S} = \langle \mathcal{C}, \vdash, \bot, \top \rangle$ *is any string generated by the following grammar:*

$$\mathbf{Agent} ::= \mathrm{tell}(C) \mid \mathrm{ask}(C) \rightarrow \mathbf{Agent} \mid \mathbf{Agent} \parallel \mathbf{Agent}$$

where $C \in |\mathcal{C}|_0$. *We will denote by* $\mathcal{A}(\mathcal{S})$ *the language of such strings. The following explicit definition is also given:*

$$\mathrm{ask}(C_1; \ldots; C_n) \rightarrow \mathbf{Agent} \;\equiv\; (\mathrm{ask}(C_1) \rightarrow \mathbf{Agent}) \parallel \cdots \parallel (\mathrm{ask}(C_n) \rightarrow \mathbf{Agent}).$$

When this will not cause confusion we will freely drop the syntactic sugar, writing C and $C_1 \rightarrow C_2$ where $\mathrm{tell}(C)$ and $\mathrm{ask}(C_1) \rightarrow \mathrm{tell}(C_2)$ are intended.

The introduction of a syntactic normal form for finite cc agents allows to simplify to subsequent semantic treatment.

Definition 4.2 (Finite cc agents: syntactic normal form.) *The transformation* η *over* $\mathcal{A}(\mathcal{S})$ *is defined, for each* $C^a, C_1^a, C_2^a, C^t \in |\mathcal{C}|_0$ *and* $A, A_1, A_2 \in \mathcal{A}(\mathcal{S})$, *as follows:*

$$\begin{aligned}
\eta(C^a \rightarrow C^t) &= \begin{cases} \mathbf{1} \rightarrow \mathbf{1} & \text{if } C^a \vdash C^t, \\ C^a \rightarrow (C^a \otimes C^t) & \text{otherwise,} \end{cases} \\
\eta(C^t) &= \mathbf{1} \rightarrow C^t, \\
\eta(C_1^a \rightarrow (C_2^a \rightarrow A)) &= \eta((C_1^a \otimes C_2^a) \rightarrow A), \\
\eta(C^a \rightarrow (A_1 \parallel A_2)) &= \eta((C^a \rightarrow A_1) \parallel (C^a \rightarrow A_2)), \\
\eta(A_1 \parallel A_2) &= \eta(A_1) \parallel \eta(A_2).
\end{aligned}$$

The following fact is easily proved.

Proposition 4.1 *The transformation* η *of* Definition 4.2 *is well defined. Furthermore, if* $A \in \mathcal{A}(\mathcal{S})$ *then* $\eta(A)$ *is of the form* $(C_1^a \rightarrow C_1^t) \parallel \cdots \parallel (C_n^a \rightarrow C_n^t)$.

Thus, by considering only agents of the form $\parallel_{i=1}^n C_i^a \rightarrow C_i^t$ we do not loose any generality. We will call elementary agents of the kind $C^a \rightarrow C^t$ *ask–tell pairs*.

Now we express the operational semantics of finite cc agents by means of rewrite rules. An agent in normal form is rewritten by applying the logical rules of the calculus modulo a structural congruence. This congruence states, intuitively, that we can regard an agent as a set of (concurrent) ask–tell pairs.

Definition 4.3 (A calculus of finite cc agents.) *Let* $\mathbf{1}_{\mathbb{A}} = \mathbf{1} \rightarrow \mathbf{1}$. *The* structural congruence *of the calculus is the smallest congruence relation* $\equiv_s$ *such that* $\langle \mathcal{A}(\mathcal{S}), \parallel, \mathbf{1}_{\mathbb{A}} \rangle_{/\equiv_s}$ *is a commutative and idempotent monoid. The reduction rules of the calculus are given in* Figure 1. *We also define the relation* $\rho_{\mathbb{A}} \subseteq \mathcal{A}(\mathcal{S}) \times \mathcal{A}(\mathcal{S})$ *given, for each* $A, A' \in \mathcal{A}(\mathcal{S})$, *by*

$$A \; \rho_{\mathbb{A}} \; A' \quad \Leftrightarrow \quad \exists n \in \mathbb{N} \,.\, A = A_1 \wedge A_n = A' \wedge A_1 \mapsto A_2 \mapsto \cdots \mapsto A_n \not\mapsto$$

Structure	$\dfrac{A_1 \equiv_s A_1' \quad A_1' \longmapsto A_2' \quad A_2' \equiv_s A_2}{A_1 \longmapsto A_2} \qquad \dfrac{A_1 \longmapsto A_1'}{A_1 \parallel A_2 \longmapsto A_1' \parallel A_2}$
Reduction	$\dfrac{C_2^a \vdash C_1^a \qquad C_1^t \vdash C_2^t}{(C_1^a \rightarrow C_1^t) \parallel (C_2^a \rightarrow C_2^t) \longmapsto (C_1^a \rightarrow C_1^t)}$
Deduction	$\dfrac{C_1^t \vdash C_2^a}{(C_1^a \rightarrow C_1^t) \parallel (C_2^a \rightarrow C_2^t) \longmapsto (C_1^a \rightarrow (C_1^t \otimes C_2^t)) \parallel (C_2^a \rightarrow C_2^t)}$
Absorption	$\dfrac{C_1^a \vdash C_2^a}{(C_1^a \rightarrow C_1^t) \parallel (C_2^a \rightarrow C_2^t) \longmapsto ((C_1^a \otimes C_2^t) \rightarrow (C_1^t \otimes C_2^t)) \parallel (C_2^a \rightarrow C_2^t)}$

Figure 1: Reduction rules for finite cc agents.

In the following we will systematically abuse the notation denoting $\mathcal{A}(\mathcal{S})_{/\equiv_s}$ simply by $\mathcal{A}(\mathcal{S})$. Consequently, every assertion concerning $\mathcal{A}(\mathcal{S})$ is to be intended *modulo structural congruence*.

Proposition 4.2 *The term-rewriting system depicted in Figure 1 is strongly normalizing. Thus the relation* $\rho_{\mathbb{A}}$ *is indeed a function* $\rho_{\mathbb{A}} : \mathcal{A}(\mathcal{S}) \rightarrow \mathcal{A}(\mathcal{S})$.

The situation here is almost identical to the one of Definition 3.2, in that we have a domain-independent strong normal form also for the present class of constraints (i.e. agents) incorporating the notion of dependency.

Definition 4.4 (Elements.) *The* elements *of* $\mathcal{A}(\mathcal{S})$ *are those which are closed under (are the fixed points of) the* inference map $\rho_{\mathbb{A}}$. *The set of elements of* $\mathcal{A}(\mathcal{S})$ *will be denoted by* $|\mathcal{A}(\mathcal{S})|$.

The strict analogy with determinate constraint systems continues with the following.

Definition 4.5 (Ask-and-tell constraint system.) *Given a simple constraint system* $\mathcal{S} = \langle \mathcal{C}, \vdash, \bot, \top \rangle$, *let* $\mathcal{A} = |\mathcal{A}(\mathcal{S})|$. *Then let* $\mathbf{0}_{\mathbb{A}}, \mathbf{1}_{\mathbb{A}} \in \mathcal{A}$, $\otimes_{\mathbb{A}} : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$, *and* $\vdash_{\mathbb{A}} \subseteq \mathcal{A} \times \mathcal{A}$ *be given, for each* $A_1, A_2 \in \mathcal{A}$, *by*

$$\begin{aligned}
\mathbf{0}_{\mathbb{A}} &= \mathbf{1} \rightarrow \mathbf{0}, & A_1 \otimes_{\mathbb{A}} A_2 &= \rho_{\mathbb{A}}(A_1 \parallel A_2), \\
\mathbf{1}_{\mathbb{A}} &= \mathbf{1} \rightarrow \mathbf{1}, & A_1 \vdash_{\mathbb{A}} A_2 &\Leftrightarrow A_1 \otimes_{\mathbb{A}} A_2 = A_1.
\end{aligned}$$

The projection operators The projection operators $\exists_\Delta^{\mathbb{A}} : \mathcal{A} \rightarrow \mathcal{A}$ *are given, for each* $\Delta \subseteq_f Vars$ *and* $A \in \mathcal{A}$, *by*

$$\exists_\Delta^{\mathbb{A}} A = \rho_{\mathbb{A}}(A_{|_\Delta}),$$

where

$$A_{|_\Delta} = \left\{ (\exists_\Delta C^a \rightarrow \exists_\Delta C^t) \;\middle|\; \begin{array}{l} (C^t \rightarrow C^a) \in A \quad \text{and} \\ ((\mathbf{1} \rightarrow \exists_\Delta C^a) \otimes_{\mathbb{A}} A) \vdash_{\mathbb{A}} (\mathbf{1} \rightarrow C^a) \end{array} \right\}.$$

Finally, let $\oplus_{\mathbb{A}} : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ *be an operator satisfying, for each* $A_1, A_2 \in \mathcal{A}$, *the following axioms:*

J_1^a. $\langle \mathcal{A}, \oplus_{\mathbb{A}}, \mathbf{0}_{\mathbb{A}} \rangle$ *is a commutative and idempotent monoid;*

J_2^a. *for each* $A_1, A_2 \in \mathcal{A}$, $A_1 \vdash_{\mathbb{A}} A_1 \oplus_{\mathbb{A}} A_2$ *and* $A_2 \vdash_{\mathbb{A}} A_1 \oplus_{\mathbb{A}} A_2$.

Again, we will denote by $\sqsubseteq_{\mathbb{A}}$ *the relation induced by* $\oplus_{\mathbb{A}}$ *over* $\mathcal{A}$: $A_1 \sqsubseteq_{\mathbb{A}} A_2$ *iff* $A_1 \oplus_{\mathbb{A}} A_2 = A_2$. *We will refer to* $\langle \mathcal{A}, \vdash_{\mathbb{A}}, \otimes_{\mathbb{A}}, \oplus_{\mathbb{A}}, \{\exists_\Delta^{\mathbb{A}}\}, \mathbf{0}_{\mathbb{A}}, \mathbf{1}_{\mathbb{A}} \rangle$ *as the* ask-and-tell constraint system *over* $\mathcal{S}$ *and* $\oplus_{\mathbb{A}}$. *We will call it* closed *iff it satisfies*

J_3^a. *for each family* $\{A_i \in \mathcal{A}\}_{i\in\mathbb{N}}$, $\bigoplus_{i\in\mathbb{N}}^a A_i = A_1 \oplus_A A_2 \oplus_A \cdots$ *exists and is unique in* $\mathcal{A}$; *moreover, associativity, commutativity, and idempotence of* $\oplus_A$ *apply to denumerable as well as to finite families of operands.*

Once you have a determinate constraint system, you also have an ask-and-tell constraint system, whose merge operator is induced as follows.

Definition 4.6 *Let* $\mathcal{S} = \langle \mathcal{C}, \vdash, \bot, \top\rangle$ *be an s.c.s., and let* $\mathcal{D} = \langle |\mathcal{C}|_0, \vdash, \otimes, \oplus, \{\exists_\Delta\}, \mathbf{0}, \mathbf{1}\rangle$ *a d.c.s. over* $\mathcal{S}$. *Let also* $\mathcal{A} = |\mathcal{A}(\mathcal{S})|$, *and let* $\hat{\oplus}_A : \mathcal{A}\times\mathcal{A} \to \mathcal{A}$ *be given by*

$$\left(\coprod_{i=1}^{n} A_i\right) \hat{\oplus}_A \left(\coprod_{j=1}^{m} B_j\right) = \rho_A\left(\coprod_{i=1}^{n}\coprod_{j=1}^{m} (A_i \hat{\oplus}_A B_j)\right),$$

where, for any two ask-tell pairs $C_1^a \to C_1^t$ *and* $C_2^a \to C_2^t$, *we define*

$$(C_1^a \to C_1^t) \hat{\oplus}_A (C_2^a \to C_2^t) = \begin{cases} \mathbf{1}_A & \text{if } C^a \vdash C^t, \\ C^a \to (C^a \otimes C^t) & \text{otherwise,} \end{cases}$$

being $C^a = C_1^a \otimes C_2^a$ *and* $C^t = C_1^t \oplus C_2^t$. *We will refer to* $\hat{\oplus}_A$ *as* the merge operator over $\mathcal{A}$ induced by $\mathcal{D}$.

Proposition 4.3 $\langle \mathcal{A}, \vdash_A, \otimes_A, \hat{\oplus}_A, \{\exists_\Delta^A\}, \mathbf{0}_A, \mathbf{1}_A\rangle$ *is an ask-and-tell constraint system. Furthermore, it is closed iff* $\mathcal{D}$ *is so.*

Notice that ask-and-tell constraint systems subsume the determinate ones, where only "tells" were considered. In fact we have $\eta(C_1) \otimes_A \eta(C_2) = \eta(C_1 \otimes C_2)$ and $\eta(C_1) \hat{\oplus}_A \eta(C_2) = \eta(C_1 \oplus C_2)$.

It is time to start showing why we are interested in this kind of constraint systems, even though for the more exciting things we have to wait until the next section, where combination of constraint domains are introduced. Ask-and-tell constraint system are needed to model approximate inference techniques which can be very useful for pattern analysis.

4.1 More pattern analysis for numeric domains

Following Section 2.1, there is another technique which is used for the analysis described in [3, 4, 2]: *relational arithmetic* [21]. This technique allows to infer constraints on the qualitative relationship of an expression to its arguments. If we take the ask-and-tell constraint system over the simple one of Section 2.1, we can describe it by a number of (concurrent) agents. Here are some of them (where $\bowtie$ ranges in $\{=, \neq, \leq, <, \geq, >\}$):

$$\begin{aligned} \text{ask}(x \bowtie 0) &\to \text{tell}((x+y) \bowtie y) \\ \text{ask}(x > 0 \wedge y > 0 \wedge x \bowtie 1) &\to \text{tell}((x * y) \bowtie y) \\ \text{ask}(x \bowtie y) &\to \text{tell}(e^x \bowtie e^y) \end{aligned}$$

An example of inference is deducing $X + 1 \leq Y + 2X + 1$ from $X \geq 0 \wedge Y \geq 0$. Notice that there is no restriction to linear constraints.

5 Powerset constraint systems

For the purpose of pattern analysis it is not necessary to represent the "real disjunction" of constraints collected through different computation paths, since we are interested in the common information only. To this end, a weaker notion of disjunction suffices. We define *powerset constraint systems*, which are instances of a well known construction, i.e., disjunctive completion[3] [10]. When this is applied to a simple constraint system $\mathcal{S}$ it yields the following.

[3] Given a poset $\langle L, \bot, \leq\rangle$, the relation $\preceq \subseteq \wp(L) \times \wp(L)$ induced by $\leq$ is given, for each $M_1, M_2 \in \wp(L)$ by $(M_1 \preceq M_2) \Leftrightarrow (\forall m_1 \in M_1 : \exists m_2 \in M_2 \,.\, m_1 \leq m_2)$. A subset $M \in \wp(L)$ is said *non-redundant* iff $\bot \notin M$ and $\forall m_1, m_2 \in M : m_1 \leq m_2 \Rightarrow m_1 = m_2$ The set of non-redundant subsets of L is denoted by $\wp_n(L)$. The function $\Omega : \wp(L) \to \wp_n(L)$ is given, for each $M \in \wp(L)$, by $\Omega(M) = M \setminus \{\, m \in M \mid m = \bot \vee \exists m' \in M \,.\, m < m' \,\}$.

Definition 5.1 (Powerset constraint system.) *Given an s.c.s.* $\langle \mathcal{C}, \vdash, \bot, \top\rangle$, *the* powerset constraint system *over* $\langle \mathcal{C}, \vdash\rangle$ *is given by* $\langle \wp_n(|\mathcal{C}|_0), \vdash_P, \otimes_P, \oplus_P, \{\exists_\Delta^P\}, \mathbf{0}_P, \mathbf{1}_P\rangle$, *where*

$$\begin{aligned} \mathbf{0}_P &= \emptyset, & \exists_\Delta^P S &= \Omega(\{\exists_\Delta C \mid C \in S\}), \\ \mathbf{1}_P &= \{1\}, & S_1 \vdash_P S_2 &\Leftrightarrow \forall C_1 \in S_1 : \exists C_2 \in S_2 \,.\, C_1 \vdash C_2, \\ S_1 \oplus_P S_2 &= \Omega(S_1 \cup S_2), & S_1 \otimes_P S_2 &= \Omega(\{\, C_1 \otimes C_2 \mid C_1 \in S_1, C_2 \in S_2 \,\}). \end{aligned}$$

With these definitions $\langle \wp_n(|\mathcal{C}|_0); \vdash_P, \mathbf{0}_P, \mathbf{1}_P, \otimes_P, \oplus_P\rangle$, is a join-complete, distributive bounded lattice. We can of course apply the powerset construction also to ask-and-tell constraint systems.

6 Combination of domains

It is well known that different data-flow analyses can be combined together. In the framework of abstract interpretation this can be achieved by means of standard constructions such as reduced product and down-set completion [8, 9]. The key point is that the combined analysis can be more precise than each of the component ones for they can mutually improve each other. However, the degree of cross-fertilization is highly dependent on the degree and quality of interaction taking place among the component domains.

We now propose a general methodology for domain combination with asynchronous interaction. The interaction among domains is asynchronous in that it can occur at any time: before, during, and after the "meet operation" in a completely homogeneous way.

This is achieved by considering ask-and-tell constraint systems built over *product* simple constraint systems. These constraint systems allow to express communication among domains in a very simple way. They also inherit all the semantic elegance of concurrent constraint programming languages, which provide the basis for their construction. Recently, a methodology for the combination of abstract domains has been proposed in [7], which is directly based onto low level actions such as *tests* and *queries*. While the approach in [7] is immediately applicable to an apparently wider range of analyses (this is one subject for further study), the approach we follow here for pattern analysis has the merit of being much more elegant.

We start with a set of simple constraint systems $\{\, \langle \mathcal{C}_i, \vdash_i, \bot_i, \top_i\rangle \mid i = 1, \ldots, n \,\}$, each expressing some properties of interest, and we wish to combine them so to: (1) perform all the analyses at the same time; and (2) have the domains cooperate to the intent of mutually improving each other. The first goal is achieved by considering the product of the simple constraint systems.

Definition 6.1 (Product of simple constraint systems.) *Given a finite family of simple constraint systems* $\mathcal{S}_i = \langle \mathcal{C}_i, \vdash_i, \bot_i, \top_i\rangle$ *for* $i = 1, \ldots, n$, *the* product *of the family is the structure given by* $\prod_{i=1}^n \mathcal{S}_i = \langle \mathcal{C}_\times, \vdash_\times, \bot_\times, \top_\times\rangle$, *where the product tokens are*

$$\mathcal{C}_\times = \{\, (c_1, \top_2, \ldots, \top_n) \mid c_1 \in \mathcal{C}_1 \,\} \cup \cdots \cup \{\, (\top_1, \ldots, \top_{n-1}, c_n) \mid c_n \in \mathcal{C}_n \,\} \cup \{\bot_\times\},$$

the product entailment is defined, for each $C \in \wp_f(\mathcal{C}_\times)$, *by*

$$\begin{array}{lcl} C \vdash_\times (c_1, \top_2, \ldots, \top_n) & \Leftrightarrow & \Pi_1(C) \vdash_1 c_1, \\ \vdots & \vdots & \vdots \\ C \vdash_\times (\top_1, \ldots, \top_{n-1}, c_n) & \Leftrightarrow & \Pi_n(C) \vdash_n c_n, \end{array}$$

where, for each $i = 1, \ldots, n$, $\Pi_i : \wp(\mathcal{C}_\times) \to \mathcal{C}_i$ *is the obvious projection mapping a set of n-tuples onto the set of i-th components. Finally,* $\bot_\times = (\bot_1, \ldots, \bot_n)$ *and* $\top_\times = (\top_1, \ldots, \top_n)$.

If you had a family of determinate constraint systems $\mathcal{D}_i$ built on top of the $\mathcal{S}_i$'s, you can easily "recycle" the merge operators $\oplus_i$ to obtain a merge operator $\oplus_\times : |\mathcal{C}_\times|_0 \times |\mathcal{C}_\times|_0 \to |\mathcal{C}_\times|_0$ which allows you to build a product d.c.s.

So, taking the product of constraint systems, we have realized the simplest form of domain combination. It corresponds to the direct product construction of [8], allowing for different analyses to be carried out at the same time. Notice that there is no communication at all among the domains.

However, as soon as we consider the ask-and-tell constraint system built over the product, we can express asynchronous communication among the domains in complete freedom. At the very least we would like to have the *smash product* among the component domains. This is realized by the agent $||_{i=1}^{n} \mathbf{0}_i \rightarrow \mathbf{0}_\times$. To say it operationally, the *smash* agent globalizes the (local) failure on any component domain. This is the only domain-independent agent we have.

Things become much more interesting when instantiated over particular constraint domains. In the CLP($\mathcal{R}$) system [16] non-linear constraints (e.g. $X = Y * Z$) are delayed (i.e. not treated by the constraint solver) until they become linear (e.g. until either Y or Z are constrained to take a single value). In standard semantic treatments this is modeled in the operational semantics by carrying over, besides the sequence of goals yet to be solved, a set of delayed constraints. Constraints are taken out from this set (and incorporated into the constraint store) as soon as they become linear.

We believe that this can be viewed in an alternative way which is more elegant, as it easily allows for taking into account the delay mechanism also in the fixpoint semantics, and makes sense from an implementation point of view. The basic claim is the following: CLP($\mathcal{R}$) has *three* computation domains: Herbrand, $\mathbb{R}$ (well, an approximation of it), and *definiteness*.

In other words, it also manipulates, besides the usual ones, constraints of the kind $X = gnd^\flat$ which is interpreted as the variable X being definitively bound to a unique value. We can express the semantics of CLP($\mathcal{R}$) (at a certain level of abstraction) with delay of non-linear constraints by considering the ask-and-tell constraint system over the product of the above three domains. In this view, a constraint of the form $X = Y * Z$ in a program actually corresponds to the agent

$$\text{ask}(Y = gnd^\flat; Z = gnd^\flat) \rightarrow \text{tell}(X = Y * Z).$$

In fact, any CLP($\mathcal{R}$) user *must* know that $X = Y * Z$ is just a shorthand for that agent! A similar treatment could be done for logic programs with delay declarations.

Obviously, this cannot be forgotten in abstract constraint systems intended to formalize correct data-flow analyses of CLP($\mathcal{R}$). Referring back to sections 2.1 and 4.1, when the abstract constraint system extracts information from non-linear constraints, i.e. $\text{ask}(Y > 0 \wedge Z > 0 \wedge Y \bowtie 1) \rightarrow \text{tell}((Y * Z) \bowtie Z)$ by relational arithmetic, you cannot simply let $X = Y * Z$ stand by itself. By doing this you would incur the risk of *overshooting* the concrete constraint system (thus loosing soundness), which is unable to deduce anything from non-linear constraints. The right thing to do is to combine your abstract constraint system with one for definiteness (by the product and the ask-and-tell construction) and considering, for example, the following agent:

$$\begin{aligned} \text{ask}(Y = gnd^\Vert; Z = gnd^\Vert) &\rightarrow \text{ask}(Y > 0 \wedge Z > 0 \wedge Y \bowtie 1) \\ &\rightarrow \text{tell}((Y * Z) \bowtie Z) \end{aligned}$$

Beware not to confuse $X = gnd^\flat$ with $X = gnd^\Vert$. The first is the *concrete one*: X is definite if and only if $X = gnd^\flat$ is entailed in the current store. In contrast, having $X = gnd^\Vert$ entailed in the current *abstract* constraint store means that X is certainly bound to a unique value in the concrete computation, but this is only a sufficient condition, not a necessary one.

Let us see another example. The analysis described in [13] aims at the compile-time detection of those non-linear constraints that will become linear at run time. This analysis is important for remedying the limitation of CLP($\mathcal{R}$) to linear constraints by incorporating powerful (and computationally complex) methods from computer algebra as the ones employed in RISC-CLP(Real) [14]. With the results of the above analysis this extension can be done in a smooth way: non-linear constraints which are guaranteed to become linear will be simply delayed, while only the other non-linear constraints will be treated with the special solving techniques. Thus, programs not requiring the extra power of these techniques will be hopefully recognized as such, and will not pay any penalties. The analysis of [13] is a kind of definiteness. One of its difficulties shows up

when considering the simplest non-linear constraint: $X = Y * Z$. Clearly X is definite if Y and Z are such. But we cannot conclude that the definiteness of Y follows from the one of X and Z, as we need also the condition $Z \neq 0$. Similarly, we would like to conclude that X is definite if Y or Z have a zero value. Thus we need approximations of the concrete values of variables (i.e. pattern analysis), something which is not captured by common definiteness analyses while being crucial when dealing with non-linear constraints. Then, just take the combination to obtain something like[4]

$$\begin{aligned} & \text{ask}(Y = gnd^\Vert \wedge Z = gnd^\Vert) \rightarrow \text{tell}(X = gnd^\Vert) \\ || \quad & \text{ask}(Y = 0; Z = 0) \rightarrow \text{tell}(X = gnd^\Vert) \\ || \quad & \text{ask}(X = gnd^\Vert \wedge Z = gnd^\Vert \wedge Z \neq 0) \rightarrow \text{tell}(Y = gnd^\Vert) \\ || \quad & \text{ask}(X = gnd^\Vert \wedge Y = gnd^\Vert \wedge Y \neq 0) \rightarrow \text{tell}(Z = gnd^\Vert) \end{aligned}$$

7 Conclusion and future work

We have shown a hierarchy of constraint systems which, both theoretically and experimentally, have several nice features. One feature we did not mention before is that proving two members of the hierarchy being one a correct approximation of the other is often quite easy.

Almost all of the ideas in this paper have been satisfactorily implemented in the CHINA analyzer [2]. The experimental results obtained with the implementation represent a strong encouragement to proceed along these lines.

In particular, we have proposed a general methodology for domain combination with asynchronous interaction. The interaction among domains is asynchronous in that it can occur at any time: before, during, and after the domains' operations in a completely homogeneous way. This is achieved by regarding semantic domains as particular kinds of (ask-and-tell) constraint systems. These constraint systems allow to express communication among domains in a very simple way. They also inherit all the semantic elegance of concurrent constraint programming languages, which provide the basis for their construction. Future work includes answering the following questions: are there variation of these ideas which are applicable also to analysis oriented towards "non-logical" properties? That is, properties which are not preserved as the computation progresses? Can we turn this constructions capturing dependence, combination, and disjunction into an algebra of constraint domains?

References

[1] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *CACM*, 26(11):832–843, 1983.

[2] R. Bagnara. On the detection of implicit and redundant numeric constraints in CLP programs. In *Proc. GULP-PRODE'94*, 1994.

[3] R. Bagnara, R. Giacobazzi, and G. Levi. Static Analysis of CLP Programs over Numeric Domains. In M. Billaud *et al.*, editors, *Actes WSA'92*, volume 81–82 of *Bigre*, pages 43–50, 1992.

[4] R. Bagnara, R. Giacobazzi, and G. Levi. An Application of Constraint Propagation to Data-Flow Analysis. In *Proc. IEEE CAIA'93*, pages 270–276, 1993. IEEE Press.

[5] P. Codognet and G. Filè. Computations, Abstractions and Constraints. In *Proc. Fourth IEEE Int'l Conference on Computer Languages*. IEEE Press, 1992.

[4] Notice that this is much more precise than the Prop formula $X \leftarrow Y \wedge Z$ [6].

[6] A. Cortesi, G. Filè, and W. Winsborough. *Prop* revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. LICS'91*, pages 322–327. IEEE Press, 1991.

[7] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming. In *Proc. POPL'94*, pages 227–239, 1994.

[8] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. POPL'79*, pages 269–282, 1979.

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

[10] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–549, 1992.

[11] E. Davis. Constraint Propagation with Interval Labels. *Artificial Intelligence*, 32:281–331, 1987.

[12] R. Giacobazzi, S. K. Debray, and G. Levi. A Generalized Semantics for Constraint Logic Programs. In *Proc. FGCS'92*, pages 581–591, 1992.

[13] M. Hanus. Analysis of nonlinear constraints in CLP($\mathcal{R}$). In D. S. Warren, editor, *Proc. ICLP'93*, pages 83–99. The MIT Press, 1993.

[14] H. Hong. RISC-CLP(Real): Logic Programming with Non-linear Constraints over the Reals. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*. The MIT Press, 1993.

[15] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. POPL'87*, pages 111–119. ACM, 1987.

[16] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) Language and System. *ACM TOPLAS'92*, 14(3):339–395, 1992.

[17] G. Janssens, M. Bruynooghe, and V. Englebert. Abstracting numerical values in CLP(H, N). In M. Hermenegildo and J. Penjam, editors, *Proc. PLILP'94*, volume 844 of *LNCS*, pages 400–414. Springer-Verlag, 1994.

[18] V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, 1993.

[19] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundation of Concurrent Constraint Programming. In *Proc. POPL'91*, pages 333–353. ACM, 1991.

[20] D. Scott. Domains for Denotational Semantics. In M. Nielsen and E. M. Schmidt, editors, *Proc. ICALP'82*, volume 140 of *LNCS*, pages 577–613. Springer-Verlag, 1982.

[21] R. Simmons. Commonsense Arithmetic Reasoning. In *Proc. AAAI-86*, pages 118–124, 1986.

if p is defined in P we use P
if p is not defined in P we use Q.

the paper is about modularity.

Tuple Inheritance: A New Kind of Inheritance for (Constraint) Logic Programming*

Juan José Moreno-Navarro Julio García-Martín Andrés del Pozo-Prieto

LSIIS, Facultad de Informática, Universidad Politécnica de Madrid

Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain

fax: +34 1 336 74 12, email: {jjmoreno,jgarcia,andres}@ls.fi.upm.es

URL: http://gedeon.ls.fi.upm.es/{~jjmoreno, ~jgarcia, ~andres}

Abstract

In this paper, we present a new form of inheritance for (constraint) logic programming. This inheritance is informally defined in the following terms: a module inherits from the other one the consequences that are not covered by itself (with respect to a fixed tuple of arguments). A computable approximation to this definition is studied, based on finite failure. In particular, we define the declarative semantics (based on Kunen's 3-valued semantics) and the operational semantics (based on constructive negation). Several examples, showing the usefulness of the proposal are presented, as well as some hints for its implementation.

Keywords: *Module Inheritance, Object Orientation, Constructive Negation.*

1 Introduction

From the software engineering point of view it is clear that modular facilities are absolutely important in a programming language. Modularity is a key feature to support a *programming-in-the-large* discipline of programming, including data abstraction, reusability support and separate compilation. Nowadays, it is widely accepted that object oriented concepts are a good basis for modularity. The different notions of inheritance provide various module composition mechanisms.

In the context of logic programming, modularity comes from several approaches. In particular, we will follow O'Keefe's approach [O'K85]: logic programs are elements of an algebra and their composition is modeled in terms of operators of the algebra.

Module composition is treated as a *meta-linguistic* mechanism using various operators to compose set of clauses: union, deletion, closure an overriding-union (see

*This research was supported in part by the spanish project TIC/93-0737-C02-02.

[O'K85, MP88, BLM92, BLM94]). All these operators have been used to model inheritance in logic programming. In object oriented programming there are two possible ways of inheriting an operation (or a state) from a module: by replacing the operation definition by a new one (overriding) o by extending its behaviour. Following the terminology of [BLM94], extension is modeled by program union ($P \cup Q$), and overriding by the operator with the same name ($P \lhd Q$), in which clauses from Q are not imported for any predicate yet defined in P.

However, in a logic program we can combine both modes of inheritance due to the nondeterministic-natured of the language. Multiple definitions for a predicate are allowed with clauses that superpose in some arguments. This is the basis for our *tuple inheritance* concept, that we denote $P \ll Q$. Instead of giving a syntactic definition of inheritance, we provide a more semantical one: for each predicate, we can inherit the part of Q that is not covered by the new definition in P. In other words, Q computations are *overridden* by P computations, and for those computations that are not in P we use Q computations instead, *extending* P behaviour.

We formalize this concept giving the declarative semantics of the new operator. In order to use a computable approximation of that "is not computed in P" we use the notion of finite failure ("it is proved that it cannot be proved"), basing our declarative semantics on Kunen's 3-valued semantics.

At a first sight, it is not very interesting because it does not matter whether an atom is true in P or in Q. It is not the case, because: i) clauses can have side effects, what supposes a different behaviour of an atom in P or Q, and ii) P definitions can provide more efficient algorithms than Q for some specific (refined) arguments. Furthermore, we define our new inheritance concept on a subset of the argument tuple for each predicate. This is specially interesting when a predicate is used to simulate a function. Although the function arguments do not change, P can define a different result. Some motivating and practical examples are given.

For the operational semantics, we also need mechanisms to decide the finite failure of a goal. *Constructive negation* [Ch88, Ch89, St91] is a good candidate. We provide a computational mechanism based on constructive negation, proving soundness and completeness results. The operational mechanism is "constructive" in the solutions of a goal that can be and cannot be overridden. The answer can provide some constraints that allow to use Q computations.

From this point of view, we think that Constraint Logic Programming (CLP) is a more natural framework to study tuple inheritance. Additionally, we provide some background for the (less developed) theory of modularity in CLP languages. Both declarative and operational semantics are developed for (modular) CLP programs.

A prototype implementation is sketched. A PROLOG implementation with constructive negation (like SEPIA-ECLIPSE from ECRC) can be used as a target language for a program transformation.

2 Motivation

In this section, we motivate the usefulness of the proposed inheritance operator. We also provide some syntax and, more briefly, recall some concepts from CLP.

A constraint logic programming language CLP($\mathcal{A}$) [JL87] depends on a structure $\mathcal{A}$ which defines the meaning of functions and constraints relation symbols of some language $\mathcal{L}$. A basic constraint is a relation $r(t_1, \ldots, t_n)$ upon terms t_i's of the structure domain (that, by abuse of notation, we still call $\mathcal{A}$), while a constraint (formula) is any formula involving other constraints and propositional connectives and quantifiers.

Giving a set of (programmed) predicate symbols $\sigma(P)$, an atom is $p\ (t_1, \ldots, t_n)$, where $p \in \sigma(P)$ and t_i's are terms. A constraint logic program P is a finite set of rules: $p\ (t_1, \ldots, t_n)\ :-\ c,\ B_1,\ \ldots,\ B_n$ where c is a constraint and $B_1, \ldots B_n (n \geq 0)$ are atoms, defining the symbols p in $\sigma(P)$.

The logical reading of a rule right hand side is the conjunction of the constraint and the atoms: $c \wedge B_1 \wedge \ldots \wedge B_n$.

The tuple inheritance operator $\ll$ is defined between two programs P and Q. For every symbol in $\sigma(P) \cup \sigma(Q)$, we specify the subsets of tuples to perform the inheritance. Without loss of generality, we can suppose that they are the first arguments in textual order. Let $M = \{\ldots, m_p, \ldots\}$ be a set of natural numbers $1 \leq m_p \leq arity\ (p)$, one for each predicate symbol $p \in \sigma(P) \cup \sigma(Q)$. The intended semantics for p in $P \ll_M Q$ can be informally defined in the following way: The atom $p\ (t_1, \ldots, t_n)$ is true if

- $p\ (t_1, \ldots, t_n)$ is true in P, or
- $p\ (t_1, \ldots, t_n)$ is true in Q but for all $s_{m_p+1}, \ldots, s_n$, $p\ (t_1, \ldots, t_{m_p}, s_{m_p+1}, \ldots, s_n)$ "is not true" in P.

Before we give the formal definition of $\ll$ let us study some examples.

Example 1: The first example is a very simple program that we will use as a running example, as well as to compare different inheritance mechanisms.

```
P : p (X, 1) :- q (X).          Q : p (Z, Y) :- r (Y).
    q (X)    :- X > 0.              r (Y)    :- Y > 0.
```

Let m_p be 1 in M. If we query $P \ll_M Q$ with the goal $p\ (X, Y)$ we expect the following two answers: yes, $X > 0$, $Y = 1$; yes. $X \leq 0$, $Y > 0$

The semantics of $P \cup Q$ contains the atoms $p\ (X, 1)$ for $X > 0$, and $p\ (Z, Y)$ for any Z, and $Y > 0$. The overriding operator $P \lhd Q$ has the following semantics: $p\ (X, 1)$ for $X > 0$. Our new operator $P \ll_M Q$ has the atoms $p\ (X, 1)$ for $X > 0$ and $p\ (X, Y)$ for $X \leq 0, Y > 0$ as semantics (see Section 3).

Now we proceed with three more elaborated examples, paradigms of useful applications of the new inheritance operator.

Example 2: The second example comes from typical object oriented programming textbooks. Vehicles are the elements of the program and we implement the function

wheels defining the number of wheels of a vehicle. Program Q establishes that all vehicles have four wheels. However, P introduces the vehicle motorcycle, as a exception to the previous function, because it has two wheels.

```
P : wheels (motorcycle, 2).          Q : wheels (X, 4).
```

A goal wheels (car, W) should answer yes, W = 4, and a goal wheels (motorcycle, W) should answer yes, W = 2. Consequently, a goal wheels (V, W) must have the following answers: yes, V = motorcycle, W = 2, and yes, V ≠ motorcycle, W = 4. This behaviour cannot be obtained neither with union nor overriding. The union says that a motorcycle has two and four wheels. The overriding operator cannot find a definition for wheels (car, W).

Tuple inheritance is made in the argument of the function, i.e. in the first argument of the relation wheels. This is absolutely coherent, because when we define a function, the new function definition can override the result and not the arguments. The simulation of functions as predicates is a natural application of tuple inheritance.

Example 3: For the third example, we apply the inheritance operator to all the arguments. One can believe that it is not useful at all because there is no difference if an atom is true in P or Q. But it is not the case if the program contains side effects. Consider, for instance, the problem of drawing a rectangle on the screen. We have a standard procedure, using basic character output encapsulated in module Q. However, in the presence of a special screen driver we can use a specific operation to plot the rectangle. The new definition is located in program P.

```
P:  draw_rectangle (B, H) :- driver (Driver), plot_rectangle (Driver, B, H).

Q:  draw_rectangle (B, H) :- write_horizontal ('-', B),
                             write_vertical ('|', B, H),
                             write_horizontal ('-', B).
```

Notice that the desired effect is not obtained by pure overriding, because we still want to draw a rectangle with Q code when the driver is not installed.

Example 4: The fourth example has some similarities with the previous one. Again we apply the inheritance operator to all the arguments. The difference is that, in program P, we provide a more efficient algorithm for some instances of the data that are generally managed by Q. Q computes the area of any polygon by dividing it into triangles, and them adding the areas of such these triangles. However, when P detects that the polygon is a rectangle, we apply the well known formula, what is more efficient than the general algorithm. We choose M to contain $m_{poligon\_area} = 2$.

```
P : polygon_area (P, A)     :- A = B * H, rectangle (P),
                               base (P, B), high (P, H).

Q : polygon_area (P, A)     :- triangle_list (P, L), sum_area (L, A).

    sum_area ([ ], 0).
    sum_area ([T | R], A)   :- A = AT + AR, triangle_area (T, AT),
                               sum_area (R, AR).
```


Notice that the intended behaviour cannot be modeled by the overriding operator $P \lhd Q$. Program union $P \cup Q$ has a similar behaviour than $P \ll_M Q$ but a call with a rectangle will provide two (equivalent) answers, the second one more costly to compute than the first one. Obviously $P \ll_M Q$ and $P \cup Q$ are not operationally equivalent, and the first operator should be more efficient than the second one. Furthermore, if we execute this program in CLP ($\Re$), we can loose some precision when Q is used and the result of P can be more accurate. Note that polygon_area is again a function definition simulated as a predicate.

3 Declarative Semantics

Following [Re88, Bu92] we believe that the meaning of any composition operator must be defined in terms of the declarative semantics. Every program operator has an associated operator between program semantics. We use the standard notion for (constraint) logic programming: An $\mathcal{A}$-interpretation I can be represented as a subset of $\mathcal{B}_\mathcal{A}$, i.e. $I \in \mathcal{P}(\mathcal{B}_\mathcal{A})$, where $\mathcal{B}_\mathcal{A} = \{p\ (d_1, \ldots, d_n) | p \in \sigma(P), d_i \in \mathcal{A}\}$. We will omit the subscript $\mathcal{A}$ whenever it does not cause ambiguity.

In [MP88] it is established that, for program composition purposes, the correct choice for the meaning of a program is the associated interpretation transformer $\mathcal{T}_P$, as shown in the following example: consider P : p (1) :- q (2). and Q : q (2). The minimal model for P is $\emptyset$ and the minimal model for Q is $\{q\ (2)\}$. Any composition of both sets cannot give the intended semantics $\{p\ (1), q\ (2)\}$ for any inheritance operator. The main reason is that the classical minimal model semantics is not valid since it is not OR-compositional.

We define the meaning of a program P, noted $[\![P]\!]$, as its associated interpretation transformer $\mathcal{T}_P$. The semantics of our inheritance operator will be defined as the composition of two interpretation transformers:

$$\mathcal{T}_{P \ll_M Q} = [\![P \ll_M Q]\!] = [\![P]\!] \odot_M [\![Q]\!] = \mathcal{T}_P \odot_M \mathcal{T}_Q$$

where $\odot_M$ is an operator in $\mathcal{P}(\mathcal{B}) \to \mathcal{P}(\mathcal{B})$. By abuse of notation we will also use $\odot_M$ to denote an operator between interpretations. Now, the transformer associated to $\mathcal{T}_{P \ll_M Q}$ is defined by:

$$\mathcal{T}_{P \ll_M Q}(I) = \mathcal{T}_P(I) \odot_M \mathcal{T}_Q(I)$$

so, we only need to define $\odot_M$ in $\mathcal{P}(\mathcal{B})$.

Due to our definition of inheritance, we cannot provide a syntactic characterization of the composition of two programs (unless we use some kind of universal quantification construct into the programs, hard to be efficiently implemented). We will recall this point in the conclusion.

Let us discuss how to define this composition operator. From the informal definition of section 2 (elements in P plus those elements in Q that are not defined in P, with respect to M), a first attempt yields to the following formal definition:

$$S_1 \odot_M S_2 = S_1 \cup \{p\ (t_1, .., t_n) \in S_2 \mid \forall\ s_{m_p+1}, .., s_n\ p\ (t_1, .., t_{m_p}, s_{m_p+1}, .., s_n) \notin S_1\}$$

Unfortunately, this definition is not computable, because the decision $p\ (\bar{t}) \notin S$ is only computable when S is finite. The previous problem is overcome with a decidable notion of "elements in Q that are not in P". A suitable definition for this concept is the set of "elements in Q that can be proved that they cannot be proved in P" (i.e. they finitely fail in P).

The usual semantics are not valid to represent this knowledge. We need to use an appropriate semantics with failure information. The same idea underlines the formal meaning of negation as failure, where the meaning of a program is given by logical consequences of its completion in a 3-valued logic [F85] and can be denotationally formalized in terms of 3-valued interpretations [Ku87]. For this reason, we extend Kunen's 3-valued semantics to CLP. The basic ideas are taken from [FBJ88] (even thought it is not directly referred to constraint logic programming) and [St91]. We reformulate them in a set-based framework. This allows us to give a definition of the semantics of $\ll_M$ in a set-based fashion as similar it is done for $\lhd$ in [Bu92]. The similarity of the definitions facilitates the comparison.

First of all, we need a different notion of interpretation.

Definition: A 3-valued interpretation is a pair $< T, F >$, where $T, F \in \mathcal{P}(\mathcal{B})$ are disjoint sets. Interpretations can be ordered in the following way:

$$< T, F > \preceq < T', F' > \quad \text{iff} \quad T \subseteq T', \text{ and } F \subseteq F'$$

Intuitively, an atom belongs to T if it is true (t), and it fails (f) if it belongs to F. Otherwise, it is undefined (u). Notice that a classic interpretation is, in particular, a 3-valued interpretation where there is no undefined atom.

Interpretations $I =< T, F >$ can be extended to arbitrary formulas in a natural manner, giving a result in $\{\mathrm{t}, \mathrm{f}, \mathrm{u}\}$: $I(A) = \mathrm{t}$ if $A \in T$ (A atom), $I(A) = \mathrm{f}$ if $A \in F$, $I(A) = \mathrm{u}$ if $A \notin T \cup F$, $I(c) = \mathrm{t}$ if $\mathcal{A} \models c$, $I(c) = \mathrm{f}$ if $\mathcal{A} \models \neg c$, and for the propositional connectives strong 3-valued interpretations are used.

The next step is to define the interpretation transformer $\mathcal{T}_P^{\mathcal{A}} : \mathcal{P}(\mathcal{B}) \to \mathcal{P}(\mathcal{B})$ (or, simply, $\mathcal{T}_P$).

Definition: $\mathcal{T}_P(I) =< T', F' >$, where:

- $p\ (\bar{t}) \in T'$ if there exists a ground instance of a clause in P, $p\ (\bar{t}) : -G$, such that $I(G) = \mathrm{t}$.
- $p\ (\bar{t}) \in F'$ if for each ground instance of a clause in P, $p\ (\bar{t}) : -G$, then $I(G) = \mathrm{f}$.

Example: Remember example 1 from Section 2. The operators for P and Q are:

$$\mathcal{T}_P(< T, F >) =< \{p\ (X,1) \mid q\ (X) \in T\} \cup \{q\ (X) \mid X > 0\}\ ,\ \{p\ (X,Y) \mid Y \neq 1\} \cup \{p\ (X,1) \mid q\ (X) \in F\} \cup \{q\ (X) \mid X \leq 0\} \cup \{r\ (X)\} >$$

$$\mathcal{T}_Q(< T, F >) =< \{p\ (Z,Y) \mid r\ (Y) \in T\} \cup \{r\ (Y) \mid Y > 0\}\ ,\ \{p\ (Z,Y) \mid r\ (Y) \in F\} \cup \{r\ (Y) \mid Y \leq 0\} \cup \{q(X)\} >$$

As usual, the semantics of a program P is defined as the least fixpoint of $\mathcal{T}_P$. Unfortunately, the operator $\mathcal{T}_P$ is not always continuous and hence its least fixpoint

may occur at any recursive ordinal, see [FBJ88]. It is continuous for logic programming (i.e. $\mathcal{A}$ is the Herbrand universe) as established in [Ku87]. In any case, the natural cut off point for computability is after ω steps, and Fitting and Ben-Jacobs [FBJ88] claim that $\mathcal{T}_P \uparrow \omega$ (noted $M_P =< T_P, F_P >$) is a natural definition of the true and failing things we can compute from a program P.

Now, we can define the semantics of $\odot_M$ in $\mathcal{P}(\mathcal{B}) \times \mathcal{P}(\mathcal{B})$.

$$< T_1, F_1 > \odot_M < T_2, F_2 >=< T_1 \cup (T_2 \cap_M F_1), F_2 \cap_M F_1 >$$

where $S_1 \cap_M S_2 = \{p\ (t_1, \ldots, t_n) \in S_1 \mid \forall\ s_{m_p+1}, \ldots, s_n\ \ p\ (t_1, \ldots, t_{m_p}, s_{m_p+1}, \ldots, s_n) \in S_2\}$

Example: Having computed $\mathcal{T}_P$ and $\mathcal{T}_Q$ for example 1, we can compute $\mathcal{T}_{P \ll Q}$.

$\mathcal{T}_{P \ll Q}(< T, F >) =$

$$\begin{array}{ll} < & \{p(Z,Y) \mid r\ (Y) \in T \wedge \forall\ W \\ & \quad p\ (Z,W) \in \{p\ (X',Y') \mid \\ & \quad\ Y' \neq 1 \vee (Y' = 1 \wedge q\ (X') \in F\}\} \cup \\ & \{p\ (X,1) \mid q\ (X) \in T\} \cup \\ & \{q\ (X) \mid X > 0\} \cup \\ & \{r(Y) \mid Y > 0\} \\ < & \{p\ (Z,Y) \mid r\ (Y) \in T \wedge q\ (Z) \in F\} \cup \\ & \{p\ (X,1) \mid q\ (X) \in T\} \cup \\ & \{q\ (X) \mid X > 0\} \cup \\ & \{r\ (Y) \mid Y > 0\} \end{array} \quad , \quad \begin{array}{l} \{p\ (Z,Y) \mid r\ (Y) \in F \wedge \forall\ W \\ \quad p\ (Z,W) \in \{p\ (X',Y') \mid \\ \quad\ Y' \neq 1 \vee (Y' = 1 \wedge q\ (X') \in F\}\} \cup \\ \{q\ (X) \mid X \leq 0\} \cup \\ \{r\ (Y) \mid Y \leq 0\} \cup \\ \{q\ (X) \mid Y \leq 0\} \\ \\ \{r\ (Y) \mid Y \leq 0\} \cup \\ \{q(X) \mid X \leq 0\} \end{array} \begin{array}{l} > = \\ \\ \\ \\ \\ \\ > \end{array}$$

$\mathcal{T}_{P \ll Q}$ has a finite fixpoint that is

$$M_{P \ll Q} =< \{p\ (X,1) \mid X > 0\} \cup \{p\ (Z,Y) \mid Y > 0, Z \leq 0\} \cup \{q\ (X) \mid X > 0\} \cup \{r\ (Y) \mid Y > 0\}\ ,\ \{p\ (Z,Y) \mid Y \leq 0, Z \leq 0\} \cup \{r\ (Y) \mid Y \leq 0\} \cup \{q(X) \mid X \leq 0\} >$$

The operator $\odot_M$ is well defined, as stated by the following theorem:

Theorem:
The operator $\odot_M$ is continuous in both arguments in $\mathcal{P}(\mathcal{B})$ for any M.

Proof:
Let $I =< T, F >, I' =< T', F' >, I'' =< T'', F'' >$. For the continuity on the first argument we need to prove that $(I \odot_M I'') \preceq (I' \odot_M I'')$ if $I \preceq I'$. It yields to prove that: $T \cup (T'' \cap_M F) \subseteq T' \cup (T'' \cap_M F')$ and $F \cap_M F'' \subseteq F' \cap_M F''$, what is obvious from set theory. The continuity on the second argument is analogous.

The result proves that $\odot_M$ preserves some properties. In particular, $\odot_M$ is continuous in the domain of continuous mappings from $\mathcal{P}(\mathcal{B}) \to \mathcal{P}(\mathcal{B})$.

4 Operational Semantics

Before we give the operational semantics of tuple inheritance, we reformulate CLP operational semantics specifying the concrete program P where we look for clauses. The rule to compute a goal G is:

$$G = c, D_1, \ldots, D_j, \ldots, D_r \vdash_P^{\mathcal{A}} c', D_1, \ldots, B_1, \ldots, B_l, \ldots, D_r$$

if $D_j = p\ (t_1, \ldots, t_n)$, there exists a (standardized apart) clause in P: $p\ (s_1, \ldots, s_n) : -c'', B_1, \ldots, B_l$, and $\mathcal{A} \models c \wedge c'' \wedge \bigwedge(s_i = t_i) \rightarrow c'$.

$c', D_1, \ldots, B_1, \ldots, B_l, \ldots, D_r$ is called a *child* of G. All the children for G forms its *derivation tree*. We will omit the superscript with the constraint domain.

A very important notion for our purpose is the concept of *frontier* of (a derivation tree for) G: a finite set of nodes in the tree such that every path from G to the leaves either contains a failure or passes through exactly one node in the set [Ch89, St91].

In order to compute when $G \vdash_{P \ll_M Q} G'$, we introduce a new operator for goals: $\forall\ \overline{X}\ \delta_P(G)$, the definitionless operator, in the vein of [MN94]. Intuitively, $\forall\ \overline{X}\ \delta_P(c_g, G)$ is true when P finitely fails for G for any value of $\overline{X}$. More formally, if $\overline{Y}$ are the free variables in c_g, G:

$$\forall\ \overline{X}\ \delta_P(c_g, G) \leftrightarrow M_P(\exists \overline{Y} c_g \wedge G\theta) = \mathbf{f} \text{ for all ground substitutions } \theta \text{ for } \overline{X}$$

We allow now to write a δ-goal $\forall\ \overline{X}\ \delta(c_g, G)$ in (intermediate) goals. Before we explain how a δ-goal is computed, we can define the rules for $\vdash_{P \ll_M Q}$ As usual, we use the notation $\Box$ to denote the empty goal formula. Let m_p be the associated inheritance arity of p in M.

$$c, D_1, \ldots, D_j, \ldots, D_r \vdash_{P \ll_M Q} c'', D_1, \ldots, G, \ldots, D_r \quad \text{if} \quad \begin{array}{l} c, D_j \vdash_P c', G, \\ \mathcal{A} \models c \wedge c' \rightarrow c'' \end{array}$$

$$c, D_1, \ldots, D_j, \ldots, D_r \vdash_{P \ll_M Q} c''', D_1, \ldots, G, \ldots, D_r$$

if
- $\forall\ X_{m_p+1}, \ldots, X_n\ \delta(p\ (t_1, \ldots, t_{m_p}, X_{m_p+1}, \ldots, X_n)) \vdash_P c', \Box$,
- $c', D_j \vdash_Q c'', G'$
- $\mathcal{A} \models c \wedge c' \wedge c'' \rightarrow c'''$

A δ-goal can be computed by adapting the technique of constructive negation. Constructive negation can be understood in a more general context than negation. It is useful to decide when a goal finitely fails (i.e. a computable approximation to undefined). The technique has been used in several frameworks: Negation in Logic Programming [Ch88, Ch89], negation in Constraint Logic Programming [St91], membership into a intensional defined set, Constraint Logic Programming with optimization, default rules for functional-logic languages [MN94], and computation of disequalities in equational logic programming.

We have only space to recall briefly the technique of constructive negation and how we apply it to our context. A $\forall\ \overline{X}\ \delta(c_g, G)$ goal is computed in the following way. Let c be the accumulated constraint when the δ-goal is computed. Let $F = \{(c \wedge c_1, B_1), \ldots, (c \wedge c_r, B_r)\}$ be a frontier of the goal $c \wedge c_g, G$ in P.

$$c, D_1, \ldots, \forall\ \overline{X}\ \delta(c_g, G), \ldots, D_r \vdash_P c, D_1, \ldots, D_{j-1}, D_{j+1}, \ldots, D_r \quad \text{if } F = \emptyset$$

$$c, D_1, \ldots, \forall\ \overline{X}\ \delta(c_g, G), \ldots, D_r \vdash_P c \wedge c'_i, D_1, \ldots, D_{j-1}, N_i, D_{j+1}, \ldots, D_r$$

where c'_i, N_i are obtained by finding a formula $(c'_1 \wedge N_1) \vee \ldots \vee (c'_l \wedge N_l)$ equivalent to the formula $\forall\ \overline{X}, \overline{Y}^1 \delta(c_1, B_1) \wedge \ldots \wedge \forall\ \overline{X}, \overline{Y}^r \delta(c_r, B_r)$, where $\overline{Y}^k$ are the variables in c_k, B_k which do not appear in c, c_g, G.

For each i $(1 \leq i \leq l)$ we have a different child in the derivation tree.

There are several concrete methods to obtain the formula $\bigvee(c'_k \wedge N_k)$:

- Chan's method [Ch88, Ch89] only applies for logic programming (CLP over equalities and disequalities in the Herbrand universe) and uses this property:

  $$\forall\ \overline{X}, \overline{Y} \delta(Z = s, G) \leftrightarrow \forall \overline{X}(Z \neq s) \vee \exists \overline{X}(Z = s \wedge \forall\ \overline{Y}\ \delta(G))$$

  where $\overline{X}$ are the non-free variables in s.

  No completeness result is provided.

- Stuckey's method [St91] applies to constraint logic programming in general. Constraint information is got from the frontier in the following way:

  $$\forall\ \overline{X}\ \delta(c, G) \leftrightarrow (\forall\ \overline{X}\ c) \vee (\forall\ \overline{X}\ \delta(c, G))$$

  A completeness theorem is proved.

- The method of [MN94] is quite similar to the previous ones although it is adapted to a different problem. The main difference is the use of a very compact constraint representation (conjunctions of disjunctions of disequalities) to minimize the number of c'_i, N_i generated.

- Drabent [Dr93] presents a different approach. Many frontiers of the derivation tree may be selected and only the constraints of such frontiers are used to compute answers. In the previous methods only one frontier is selected but whole goal bodies of the frontier are used. This yields to subgoals that may contain δ-subgoals to be resolved by further derivation steps. [Dr93] claims that the new method may be more efficient than the previous one. There are also soundness and completeness results.

Example: The computation of the goal $p\ (X, Y)$ in our running example is got in the following steps:

It is clear that (1) $p\ (X, Y) \vdash_P Y = 1, q\ (X) \vdash_P Y = 1, X > 0$

what supposes $p\ (X, Y) \vdash_{P \ll_M Q} Y = 1, X > 0$

For the other derivation in $P \ll_M Q$ we need to compute the δ-goal $\forall\ Y\ \delta(p\ (X, Y))$ in P. Derivation (1) also gives us a frontier for $p\ (X, Y)$, $\{Y = 1, X > 0\}$ which complements to the formula $Y \neq 1 \vee X \leq 0$. If we choose the second part of the formula as c, N we have:

$$\forall\ Y\ \delta(p\ (X, Y)) \vdash_P X \leq 0$$

As we can derive $X \leq 0, p\ (X, Y) \vdash_Q X \leq 0, Y > 0$

we can conclude $p\ (X, Y) \vdash_{P \ll_M Q} X \leq 0, Y > 0$

and we compute all the solutions in $M_{P \ll_M Q}$.

Now, we are in a position to establish soundness and completeness results. We assume that the rules for $\vdash$ are applied in a fairly consistent way in the sense of [St91]. We have only space to include an informal sketch of the proofs.

Theorem: **Soundness**

If $c, G \vdash_{P \ll_M Q} c', G'$ then $M_{P \ll_M Q}(c \wedge G) = M_{P \ll_M Q}(c' \wedge G')$.

Proof sketch: We proceed by induction on the structure of the goal G. The most interesting case is the base case, where G is an atom A. The rule $c, A \vdash_P c', A'$ implies $c, A \vdash_{P \ll_M Q} c', A'$ is obviously correct, because corresponds to "$T_P \cup \ldots$" in the definition of $\odot_M$. The second rule imposes (i) $c'', A \vdash_Q c', A'$, what implies that (a ground instance of) $A \in T_Q$, and (ii) $c, \forall\, \overline{X}\ \delta(A) \vdash_P c'', \Box$, what implies $A \in F_P$ by soundness of constructive negation [St91].

Theorem: **Completeness**

Let c, G be a goal with free variables $\overline{X}$. If $M_{P \ll_M Q}(\exists\, \overline{X}\ c \wedge G) = \mathsf{t}$ then there exists a constraint answer c' such that $c, G \vdash_{P \ll_M Q} c', \Box$ and $\mathcal{A} \models c \rightarrow c'$.

Proof sketch: Double induction on the structure of G and the step n on the interpretation transformer $\mathcal{T}_{P \ll_M Q} \uparrow n$ where we find the elements of (a ground instance of) G. The proof combines the completeness of constructive negation [St91] with a case analysis of the definition of $\odot_M$.

5 Implementation

A prototype implementation has been constructed by translating modules to SEPIA-ECLIPSE Prolog. As far as we know, this Prolog version provides the only existing implementation of constructive negation. The transformation is carried out in the following way. For each $p \in \sigma(P) \cup \sigma(Q)$ we define:

$\mathsf{p}\ (\mathsf{X}_1, \ldots, \mathsf{X}_n)$:- $\mathsf{p}_P\ (\mathsf{X}_1, \ldots, \mathsf{X}_n)$.
$\mathsf{p}\ (\mathsf{X}_1, \ldots, \mathsf{X}_n)$:- not $(\mathsf{p}_P\ (\mathsf{X}_1, \ldots, \mathsf{X}_{m_p}, \mathsf{Y}_{m_p+1}, \ldots, \mathsf{Y}_n))$, $\mathsf{p}_Q\ (\mathsf{X}_1, \ldots, \mathsf{X}_n)$.

where p_P, p_Q rename p in P and Q respectively, and the Y_i's are new fresh variables.

However, it is clear that a direct implementation should be more efficient. For instance, if all p-calls are ground, a sound and complete implementation will be the following:

$$\mathsf{p}\ (\overline{X})\ \text{:-}\ \mathsf{p}_P\ (\overline{X}), !. \qquad \mathsf{p}\ (\overline{X})\ \text{:-}\ \mathsf{p}_Q\ (\overline{X}).$$

A specific implementation, like a WAM modification, can use this technique when the arguments are ground.

6 Conclusion

We have introduced and formalized a new inheritance operator which combines extension and overriding for goals with variables in a natural way. This new operator

allows for a fully treatment of inheritance in a *logic* framework. However, tuple inheritance does not replace the other inheritance operators (union and overriding) but complements them.

The CLP formulation also provides a background for the study of modularity in CLP programs. A new use of the technique of constructive negation is found, what enforces our believe that the technique goes beyond negation in logic programming.

The meaning of our proposal is given in terms of the declarative semantics, instead of the syntactic characterization of other modular operations. The syntactic characterization is possible if δ-goals are allowed in clause bodies. We assume that the programs are in normal form, i.e. all the clause heads are written as $p\ (X_1, \ldots, X_n)$, where the X_i's are distinct variables and the constraint of the clause contains the equalities of X_i's with the original head's terms. Suppose we have the following programs:

$$\begin{array}{ll} \mathsf{P}\text{: } \mathsf{p}\ (\overline{\mathsf{X}})\ \text{:-}\ \mathsf{G}_1. & \mathsf{Q}\text{ : } \mathsf{p}\ (\overline{\mathsf{X}})\ \text{:-}\ \mathsf{G'}_1. \\ \vdots & \vdots \\ \mathsf{p}\ (\overline{\mathsf{X}})\ \text{:-}\ \mathsf{G}_k. & \mathsf{p}\ (\overline{\mathsf{X}})\ \text{:-}\ \mathsf{G'}_r. \end{array}$$

We can joint P with the modified Q clauses:

$$\mathsf{p}\ (\overline{\mathsf{X}})\ \text{:-}\ \forall\, \overline{\mathsf{Z}}^i, \overline{\mathsf{Y}}\ \delta_P\ (\mathsf{G}_1), \ldots, \forall\, \overline{\mathsf{Z}}^i, \overline{\mathsf{Y}}\ \delta_P\ (\mathsf{G}_k), \mathsf{G'}_i.$$

where $\overline{\mathsf{Y}} = \mathsf{Y}_{m_p+1}, \ldots, \mathsf{Y}_n$ are new fresh variables that replace the corresponding X's in each G_i, and $\overline{\mathsf{Z}}^i$ are the free variables in $\mathsf{G'}_i$. In our running example, the unique rule for predicate p will be:

$$\mathsf{p}\ (\mathsf{Z}, \mathsf{Y})\ \text{:-}\ \forall\, \mathsf{W}\ \delta_P\ (\mathsf{q}\ (\mathsf{Z})), \mathsf{r}\ (\mathsf{Y}).$$

It can be proved that the semantics of this program coincides with the semantics we have developed.

This transformation also gives us a hint to treat some simpler cases. If the G_i's have no free variables we can replace the δ-goals by adequate constraints. Informally, the *complement* of head terms of P clauses are computed and they are used later in the constraint of the modified Q predicate. The same idea is used into the transformational approach to negation from [BMPT90]. We can modify our example by replacing Q rule for p by the rules:

$$\mathsf{p}\ (\mathsf{Z}, \mathsf{Y})\ \text{:-}\ \mathsf{q'}\ (\mathsf{Z}), \mathsf{r}\ (\mathsf{Y}). \qquad \mathsf{q'}\ (\mathsf{Z})\ \text{:-}\ \mathsf{Z} \leq 0.$$

A concrete implementation can use this transformation technique whenever possible, the cut trick to be executed dynamically and constructive negation only when it is absolutely necessary.

It is worth to mention that our construction is also valid for normal programs (i.e. logic programs with negation), because the semantics can handle negative information by means of finite failures.

As a future work, we plan to experiment further with the implementation of constructive negation and to apply these ideas to functional-logic languages, because functions are a very natural framework for tuple inheritance.

References

[BMPT90] R. Barbuti, D. Mancarella, D. Pedreschi, F. Turini. A Transformational Approach to Negation in Logic Programming *Journal of Logic Programming*, 8(3):201-228, 1990.

[BLM90] A. Brogi, E. Lamma, P. Mello. Inheritance and Hypothetical Reasoning in Logic Programming *Proceedings of 9th European Conference on Artificial Intelligence, Pitman*, 1990, pp. 105-110.

[BLM92] A. Brogi, E. Lamma, P. Mello. Compositional Model-theoretic Semantics for Logic Programs. *New Generation Computing*, 11(1):1-21,1992.

[BLM93] A. Brogi, E. Lamma, P. Mello. Composing Open Logic Programs *Journal of Logic and Computation*, 4(4):417-439, 1993.

[BLM94] M. Bugliesi, E. Lamma, P. Mello. Modularity in Logic Programming *Journal of Logic Programming*, vol. 19 & 20, 1994, pp. 443-502.

[Bu92] M. Bugliesi. A Declarative View of Inheritance in Logic Programming *Proc. Joint International Conference and Symposium on Logic Programming, The MIT Press*, 1992, pp. 113-130.

[Ch88] D. Chan. Constructive Negation Based on the Complete Database *Proc. ICLP'89, The MIT Press*, 1988, 111-125.

[Ch89] D. Chan. An Extension of Constructive Negation and its Application in Coroutining *Proc. NACLP'89, The MIT Press*, 1989, 477-493.

[Dr93] W. Drabent. What is Failure? An Approach to Constructive Negation to appear in Acta Informatica.

[F85] M. Fitting. A Kripke/Kleene Semantics for Logic Programs *Journal of Logic Programming* 2 (4), 1985, pp. 295-312.

[FBJ88] M. Fitting, M. Ben-Jacob. Stratified and Three-valued Logic Programming Semantics *Conf. and Symp. on Logic Programming*, 1988, pp. 1054-1069.

[JL87] J. Jaffar, J.L. Lassez. Constraint Logic Programming *Procs. 14th ACM Symp. on Princ. of Prog. Lang.*, 1987, pp. 114-119.

[Ku87] K. Kunen. Negation in Logic Programming *Journal of Logic Programming*, 4, 1987, pp. 289-308.

[MP88] P. Mancarella, D. Pedreschi. An Algebra of Logic Programs *Proc. 5th Int. Conference on Logic Programming, The MIT Press*, 1988, pp. 1006-1023.

[MP90] L. Monteiro, A. Porto. A Transformational View of Inheritance in Logic Programming *Proc. 7th Int. Conference on Logic Programming, The MIT Press*, 1990, pp. 481-494.

[MP91] L. Monteiro, A. Porto. Syntactic and Semantic Inheritance in Logic Programming *Workshop on Declarative Programming,Springer Verlag*, 1991.

[MN94] J.J. Moreno-Navarro. Default Rules: An Extension of Constructive Negation for Narrowing-based Languages. *ICLP'94, The MIT Press*, pp. 535-554.

[O'K85] R. O'Keefe. Towards an Algebra for Constructing Logic Programs *IEEE Symposium on Logic Programming.*1985, pp. 152-160.

[Re88] U. Reddy. Objects as Closures: Abstract Semantics of Object Oriented Languages. *ACM - Lisp and Functional Programming*, 1988, pp. 289-297.

[St91] P. Stuckey. Constructive Negation for Constraint Logic Programming *Proc. IEEE Symp. on Logic in Computer Science, IEEE Comp. Soc. Press*, 1991.

ANALYSIS

Declarative diagnosis revisited

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

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

Abstract

We extend the declarative diagnosis methods to the diagnosis w.r.t. computed answers. We show that absence of uncovered atoms implies completeness for a large class of programs. We then define a top-down diagnoser, which uses one oracle only, does not require to determine in advance the symptoms and is driven by a (finite) set of goals. Finally we tackle the problem of effectivity, by introducing (finite) partial specifications. We obtain an effective diagnosis method, which is weaker than the general one in the case of correctness, yet can efficiently be implemented in both a top-down and in a bottom-up style.

Keywords: Declarative diagnosis, Verification, Semantics, Debugging

1 Introduction

The diagnosis problem can formally be defined as follows. Let P be a program, $[\![P]\!]$ be the behavior of P w.r.t. the observable property α, and $\mathcal{I}$ be the specification of the *intended* behavior of P w.r.t. α. The diagnosis consists of comparing $[\![P]\!]$ and $\mathcal{I}$ and determining the "errors" and the program components which are sources of errors, when $[\![P]\!] \neq \mathcal{I}$. The formulation is parametric w.r.t. the property considered in the specification $\mathcal{I}$ and in the actual behavior $[\![P]\!]$. *Declarative diagnosis* [13, 12, 10, 8] is concerned with model-theoretic properties. The specification is the intended

declarative semantics (the least Herbrand model in [13] and the set of atomic logical consequences in [8]).

Abstract diagnosis [4, 5] is a generalization of declarative diagnosis, where we consider operational properties, i.e., *observables* (an observable is any property which can be extracted from a goal computation, i.e., *observables are abstractions of SLD-trees*). An example of a useful observable is *computed answers*. The diagnosis w.r.t. computed answers is expected to be more precise than the declarative diagnoses in [13] and [8], which can be reconstructed in terms of the observables *ground instances of computed answers* and *correct answers* respectively [5]. The semantics involved in the diagnosis w.r.t. computed answers is the *s*-semantics [6, 7, 2], which models exactly the process of computing answers.

In this paper we first extend to computed answers the declarative diagnosis methods based on the detection of incorrect clauses and uncovered atoms (Section 3). The good news is that absence of uncovered atoms implies completeness, for a large class of interesting programs (acceptable programs).

We then define in Section 4 a top-down diagnoser, which uses one oracle only, does not require to determine in advance the symptoms and is driven by a (finite) set of goals (most general atomic goals).

Finally in Section 5 we tackle the problem of effectivity, by introducing (finite) partial specifications. We obtain an effective diagnosis method, which is weaker than the general one in the case of correctness, yet can efficiently be implemented in both a top-down and in a bottom-up style.

2 The semantics modeling computed answers

The *s*-semantics [6, 7, 2] is defined on interpretations consisting of sets of possibly non-ground atoms. For every program P, the *s*-semantics can be characterized as the least fixpoint of the operator T_P:

$$T_P(I) = \{A\theta \in B_P \mid \exists A :- B_1, \ldots, B_n \in P, \{B'_1, \ldots, B'_n\} \subseteq I, \exists \theta = mgu((B_1, \ldots, B_n), (B'_1, \ldots, B'_n))\}$$

where B_P is the set of (possibly non-ground) atoms of L_P modulo variance and I is a subset of B_P. The same denotation can be obtained in a top-down way, by considering the answers computed for "most general atomic goals", as shown by the following definition.

$$\mathcal{O}_P = \{p(X_1, \ldots, X_n)\theta \in B_P \mid X_1, \ldots, X_n \text{ are distinct variables}, ? - p(X_1, \ldots, X_n) \xrightarrow{\theta} \square\}.$$

3 Diagnosis w.r.t. computed answers: basic definitions and results

The following Definitions 3.1 and 3.2 extend to diagnosis w.r.t. computed answers the definitions given in [13, 8, 10] for declarative diagnosis.

In the following $\mathcal{I}$ is the specification of the intended *s*-semantics of P.

Definition 3.1

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

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

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

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

Definition 3.2

i. An incorrectness symptom *is an atom* A *such that* $A \in \mathcal{O}(P)$ *and* $A \notin \mathcal{I}$.

ii. An incompleteness symptom *is an atom* A *such that* $A \in \mathcal{I}$ *and* $A \notin \mathcal{O}(P)$.

Note that a totally correct program has no incorrectness and no incompleteness symptoms. Our incompleteness symptoms are related to the insufficiency symptoms in [8], which are defined by taking $\mathsf{gfp}(T_P)$ instead of $\mathcal{O}(P) = \mathsf{lfp}(T_P)$ as program semantics. The two definitions, even if different, turn out to be the same for the class of programs we are interested in (see Section 3). Ferrand's choice is motivated by the fact that $\mathsf{gfp}(T_P)$ is related to finite failures. The approach of using two different semantics for reasoning about incorrectness and incompleteness has been pursued in [9], leading to an elegant uniform (yet non-effective) characterization of correctness and completeness.

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

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

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

Definition 3.4 *An atom A is* uncovered *if $A \in \mathcal{I}$ and $A \notin T_P(\mathcal{I})$.*

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

It is worth noting that checking the conditions of Definitions 3.3 and 3.4 requires one application of T_P to $\mathcal{I}$, while the detection of symptoms according to Definition 3.2 would require the construction of $\mathcal{O}(P)$ and therefore a fixpoint computation. As we will show in the following, the detection of bugs can be based on Definitions 3.3 and 3.4, while this is not the case for Definition 3.2.

The following theorems are instances of the corresponding theorems proved in [5] for abstract diagnosis, where they are given for a class of properties called *s-observables* (computed answers is an s-observable [3]).

The first theorem shows the relation between partial correctness (Definition 3.1) and absence of incorrect clauses (Definition 3.3).

Theorem 3.5 *If there are no incorrect clauses in P, then P is partially correct (hence there are no incorrectness symptoms). The converse does not hold.*

The theorem shows the feasibility of a diagnosis method for incorrectness based on the comparison between $\mathcal{I}$ and $T_P(\mathcal{I})$. Note that the second part of the theorem asserts that there might be incorrect clauses even if there are no incorrectness symptoms. In other words, if we just look at the semantics of the program, some incorrectness bugs can be "hidden" (because of an incompleteness bug).

As in the case of declarative debugging, handling completeness turns out to be more complex, since some incompletnesses cannot be detected by comparing $\mathcal{I}$ and $T_P(\mathcal{I})$. The following proposition shows that we cannot base the diagnosis of incompleteness on the detection of uncovered atoms.

Proposition 3.6 *There exist a program P and a specification $\mathcal{I}$, such that*

i. there are no uncovered atoms in P,

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

However, the following theorem shows that the diagnosis of incompleteness can be based on Definition 3.4 if the operator T_P has a unique fixpoint.

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

Note that, if T_P has a unique fixpoint, $\mathsf{lfp}(T_P) = \mathsf{gfp}(T_P)$. Hence our incompleteness symptoms are exactly the insufficiency symptoms in [8].

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

Corollary 3.8 *Assume T_P has a unique fixpoint. Then P is totally correct w.r.t. $\mathcal{I}$, if and only if there are no incorrect clauses and uncovered atoms.*

The requirement on T_P seems to be very strong. However, this property holds for a large class of programs, i.e., for *acceptable programs* as defined in [1]. Acceptable programs are the left-terminating programs, i.e., those programs for which the *SLD*-derivations of ground goals (via the leftmost selection rule) are finite. Most interesting programs are acceptable (all the pure PROLOG programs in [14] are reported in [1] to be acceptable). The same property holds for most of the wrong versions of acceptable programs, since most "natural" errors do not affect the left-termination property. One relevant technical property of acceptable programs is that the ground immediate consequences operator has a unique fixpoint [1]. The same property holds for the s-semantics operator T_P.

Theorem 3.9 (fixpoint uniqueness) *Let P be an acceptable program. Then $T_P{\uparrow}\omega$ is the unique fixpoint of T_P.*

The theorem is proved in [4] for all the "immediate consequences" operators corresponding to s-observables. Note that the same result applies to declarative diagnosis as well.

The overall diagnosis method for acceptable programs is then given by the following corollary.

Corollary 3.10 *Assume P is an acceptable program. Then P is totally correct w.r.t. $\mathcal{I}$, if and only if there are no incorrect clauses and uncovered atoms.*

Example 3.11 Consider the acceptable program P of figure 1, which is an "ancestor" program with a wrong clause (`ancestor(X,Y) :- parent(Y,X).` instead of `ancestor(X,Y) :- parent(X,Y).`) and missing database tuples.

$$\mathcal{I} = \{\, parent(terach, abraham),\ parent(abraham, isaac),\ ancestor(terach, abraham),\ ancestor(terach, isaac),\ ancestor(abraham, isaac)\}.$$

$$T_P(\mathcal{I}) = \{\, ancestor(abraham, terach),\ ancestor(isaac, abraham),\ ancestor(terach, isaac)\}.$$

The diagnosis delivers the following result:

i. the clause `ancestor(X,Y) :- parent(Y,X).` is incorrect on $ancestor(abraham, terach)$ and $ancestor(isaac, abraham)$.

ii. the atoms $parent(terach, abraham)$, $parent(abraham, isaac)$, $ancestor(terach, abraham)$ and $ancestor(abraham, isaac)$ are uncovered.

Note that $\mathcal{O}(P) = \{\}$. Hence there are no incorrectness symptoms, even if there is an incorrect clause. Note also that the atom $ancestor(terach, isaac)$ is not uncovered, even if it is an incompleteness symptom.

The example is intended to show the relation among the various concepts involved in the diagnosis and does not use the features of the s-semantics (which turns out to be a Herbrand interpretation). ∎

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

Figure 1: A wrong acceptable program

4 The oracle and the "top-down" diagnosis

The "bottom-up" diagnosis is based on Corollary 3.10 and requires the application of T_P to the intended s-semantics $\mathcal{I}$. Hence $\mathcal{I}$ has to be specified in an extensional way. We are not concerned, for the time being, with the problem of effectivity (i.e., finiteness of $\mathcal{I}$). Rather we are concerned with the problem of specifying $\mathcal{I}$ by means of an *oracle*, as first suggested in [13]. The oracle is usually implemented by querying the user. Several oracles have been used in declarative debugging (see the discussion in [11]). We will use one oracle only, directly related to the property we are concerned with, namely computed answers.

Definition 4.1 (oracle) *Let G be a goal.*
$\mathcal{A}(G) = \{G\theta \mid G \text{ computes } \theta \text{ according to the intended } s\text{-semantics}\}$.

Once we have the oracle, we can define the *oracle simulation*, again following [13]. The oracle simulation allows us to express in a compact way new top-down diagnosis conditions. The oracle simulation performs one step of goal rewriting by using the program clauses and then gets the answers for the resulting goal from the oracle.

Definition 4.2 (oracle simulation) *Let G be an atomic goal and P be a set of definite clauses.*

$$\begin{aligned}\mathcal{S}(G,P) = \{G\theta_1\theta_2 \mid\ & \exists c = A :- B_1,\ldots,B_n \in P,\\ & \exists\theta_1 = mgu(G,A),\\ & \exists\theta_2, (B_1,\ldots,B_n)\theta_1\theta_2 \in \mathcal{A}((B_1,\ldots,B_n)\theta_1)\}\end{aligned}$$

Note that the elements of the sets computed by $\mathcal{A}$ and $\mathcal{S}$ are equivalence classes w.r.t. variance, as was the case for the domain of the s-semantics. The following two theorems justify the top-down diagnosis.

Theorem 4.3 *The clause $c \in P$ is incorrect on the atom $p(X_1,\ldots,X_n)\theta$ if and only if $p(X_1,\ldots,X_n)\theta \in \mathcal{S}(p(X_1,\ldots,X_n),\{c\})$ and $p(X_1,\ldots,X_n)\theta \notin \mathcal{A}(p(X_1,\ldots,X_n))$.*

Theorem 4.4 *The atom $p(X_1,\ldots,X_n)\theta$ is uncovered if and only if*
$p(X_1,\ldots,X_n)\theta \in \mathcal{A}(p(X_1,\ldots,X_n))$ and
$p(X_1,\ldots,X_n)\theta \notin \mathcal{S}(p(X_1,\ldots,X_n),P)$.


```
incorrect(G :- B) :- userdefined(G),
                     clause(G,B),
                     answer(B),
                     freeze(G,G1),
                     not(answer(G),G = G1).
uncovered(A) :-      userdefined(A),
                     answer(A),
                     freeze(A,A1),
                     not(clause(A,B),answer(B),A = A1).
```

Figure 2: The top-down diagnosis meta-program

The proofs of Theorems 4.3 and 4.4 are based on the properties of the s-semantics, which relate fixpoint bottom-up computations to top-down refutations for most general atomic goals. The same properties allow us to define systematic diagnosis algorithms which do not need symptoms as inputs. The PROLOG meta-program in Figure 2 is an adaptation of the simplest possible declarative debugger in [11].

The oracle `answer` nondeterministically instantiates its argument. The search for incorrect clause instances and uncovered atoms is driven by the most general atomic goals, represented by unit clauses of the form `userdefined(p(X1,...,Xn)).`, for any predicate p occurring in the program P. The properties of the s-semantics guarantee that we can detect all the incorrect clause instances and the uncovered atoms (for acceptable programs), by just looking at the behaviors for a finite number of atomic goals.

The diagnosis meta-program can be extended to achieve a better performance and to improve the calls to the oracle. Most of the techniques presented in [11] are applicable. However, performance issues are outside the scope of this paper.

Let us finally note that our formalization of diagnosis based on the s-semantics is not subject to the theoretical limitations proved by Ferrand [8] for his construction based on the atomic logical consequences semantics. The problem is the following. An incorrect clause instance $A :- B$ may have an instance $(A :- B)\theta$ which is not incorrect. This should be reflected by the fact that $incorrect(A :- B)$ is in the denotation of the diagnoser, while $incorrect((A :- B)\theta)$ is not. This is not possible if the denotation is the non-ground semantics in [8], since it is closed under instantiation. On the contrary, if we choose the s-semantics the problem does not arise.

5 Diagnosis with partial specifications

The diagnosis cannot effectively be based on the conditions given above, unless the intended s-semantics is finite. In fact, if this is not the case,

- the bottom-up diagnosis is unfeasible, since $\mathcal{I}$ is infinite and
- the top-down diagnosis is unfeasible, because the oracle may return infinite answers to some queries.

This is true also for those diagnosis algorithms which are based on a ground semantics or are driven by the symptoms. As a matter of fact, the assumption in [13] on the oracle returning a finite number of answers is too strong. The problem can only be solved if we have the ability to handle finite approximations of the intended semantics. One solution can be found within the abstract diagnosis framework in [4, 5], where we are able to cope with abstractions of the observables (according to abstract interpretation theory).

Here we propose a different solution, where we approximate the intended behavior by a (finite) partial specification. The specification of the intended behavior $\mathcal{I}$ is approximated by a partial specification, which is a pair $(\mathcal{I}^+, \mathcal{I}^-)$, where

- $\mathcal{I}^+$ is the (positive) partial specification of the answers computed by P for most general atomic goals, i.e., $\mathcal{I}^+$ is a finite subset of $\mathcal{I}$,
- $\mathcal{I}^-$ is the (negative) partial specification of the answers not computed by P for most general atomic goals, i.e., $\mathcal{I}^-$ is a finite subset of $\overline{\mathcal{I}}$.

We denote by $\overline{\mathcal{I}}$ the complement of $\mathcal{I}$. Note that the relation $\mathcal{I}^+ \subseteq \overline{\mathcal{I}^-}$ must hold. The following definition generalizes partial correctness and completeness to the case of partial specifications.

Definition 5.1

i. P *is* partially p-correct *w.r.t.* $(\mathcal{I}^+, \mathcal{I}^-)$, *if* $\mathcal{O}(P) \subseteq \overline{\mathcal{I}^-}$.

ii. P *is* p-complete *w.r.t.* $(\mathcal{I}^+, \mathcal{I}^-)$, *if* $\mathcal{I}^+ \subseteq \mathcal{O}(P)$.

The rationale behind Definition 5.1 is clearly related to the fact that the specification is partial. In a partially p-correct program, for any goal G, there is no computed answer θ, which we know to be wrong ($G\theta \in \mathcal{I}^-$). On the other hand, in a p-complete program all the answers that we know to be correct ($G\theta \in \mathcal{I}^+$) have to be computed answers. Note also that definition 5.1 is derived from Definition 3.1 by taking $\overline{\mathcal{I}^-}$ and $\mathcal{I}^+$ as specifications to be used for correctness and completeness respectively.

Positive and negative specifications have been used in [9] with the aim of separately modeling the behavior w.r.t. incorrectness and incompleteness. $\mathcal{I}^+$ and $\mathcal{I}^-$ are not partial specifications, rather they are specifications of the (complete) intended $\mathsf{lfp}(T_P)$ and of the (complete) intended $\overline{\mathsf{gfp}(T_P)}$. The derived definitions and results are completely different from ours. In particular, $\overline{\mathcal{I}^-}$ is used for completeness and $\mathcal{I}^+$ is used for correctness.

The following definitions, given in terms of the T_P operator, generalize the definitions of incorrect clause and uncovered atom to the case of partial specifications.

Definition 5.2 *If there exists an atom* A *such that* $A \notin \overline{\mathcal{I}^-}$ *and* $A \in T_{\{c\}}(\overline{\mathcal{I}^-})$, *then the clause* $c \in P$ *is* p-incorrect *on* A.

Definition 5.3 *An atom* A *is* p-uncovered *if* $A \in \mathcal{I}^+$ *and* $A \notin T_P(\mathcal{I}^+)$.

The following theorem shows the relation between partial p-correctness and absence of p-incorrect clauses.

Theorem 5.4 *If there are no p-incorrect clauses in* P, *then* P *is partially p-correct. The converse does not hold.*

Theorem 5.4 would allow us to check partial p-correctness, by just checking that there are no p-incorrect clauses. However, we cannot base an effective diagnosis method on the detection of p-incorrect clauses, since Definition 5.2 is given in terms of $\overline{\mathcal{I}^-}$, which is not part of the partial specification (and is usually infinite). Some of the p-incorrect clauses can be determined by choosing $\mathcal{I}^+$ as an approximation of $\overline{\mathcal{I}^-}$, as shown by the following theorem.

Theorem 5.5 *If there exists a clause* c *in* P *and an atom* A, *such that* $A \in T_{\{c\}}(\mathcal{I}^+) \cap \mathcal{I}^-$, *then* c *is p-incorrect on* A. *The converse does not hold.*

Theorem 5.5 leads to a complete diagnosis method for partial p-correctness, only if the specification is indeed complete, i.e., if $\mathcal{I}^+ = \overline{\mathcal{I}^-}$.

Corollary 5.6 *If* $\mathcal{I}^+ = \overline{\mathcal{I}^-}$ *and there are no clauses* c *in* P *and atoms* A *such that* $A \in T_{\{c\}}(\mathcal{I}^+) \cap \mathcal{I}^-$, *then* P *is partially p-correct.*

Let us consider now the diagnosis of p-completeness. As was the case for the diagnosis of completeness, the diagnosis can be based on T_P, only if the operator T_P has a unique fixpoint.

Theorem 5.7 *Assume* T_P *has a unique fixpoint. If there are no p-uncovered atoms, then* P *is p-complete w.r.t.* $(\mathcal{I}^+, \mathcal{I}^-)$. *The converse does not hold.*

It is worth noting that the existence of a p-uncovered atom does not necessarily mean that there is something missing from the program. In fact, an atom in $\mathcal{I}^+$ might not be in $T_P(\mathcal{I}^+)$ just because $\mathcal{I}^+$ is partial, i.e., it cannot be derived by T_P because some of the correct premises are missing from $\mathcal{I}^+$. Hence, the overall partial diagnosis may return a subset of the incorrect clauses and a superset of the real uncovered atoms.

Let us now move to the top-down diagnosis with partial specifications. The definition is given in terms of two oracles, which can be implemented either by querying the user or by querying the positive and negative specifications, since they are finite and can be defined extensionally.

```
pincorrect(G :- B) :- nanswer(G),
                      freeze(G,G1),
                      clause(G,B),
                      panswer(B),G = G1.
puncovered(A) :-      panswer(A),
                      freeze(A,A1),
                      not(clause(A,B),panswer(B),A = A1).
```

Figure 3: The top-down diagnosis meta-program for partial specifications

Definition 5.8 (positive oracle) *Let G be a goal.*
$\mathcal{A}^+(G) = \{G\theta \mid G \text{ is intended to compute } \theta\}$.

Definition 5.9 (negative oracle) *Let G be a goal.*
$\mathcal{A}^-(G) = \{G\theta \mid G \text{ is intended not to compute } \theta\}$.

We only need the positive oracle simulation.

Definition 5.10 (positive oracle simulation) *Let G be an atomic goal and P be a set of definite clauses.*

$$\mathcal{S}^+(G,P) = \{G\theta_1\theta_2 \mid \exists c = A :- B_1,\ldots,B_n \in P, \exists\theta_1 = mgu(G,A), \exists\theta_2, (B_1,\ldots,B_n)\theta_1\theta_2 \in \mathcal{A}^+((B_1,\ldots,B_n)\theta_1)\}$$

The following two theorems justify the top-down diagnosis.

Theorem 5.11 *The clause $c \in P$ is p-incorrect on the atom $p(X_1,\ldots,X_n)\theta$ if $p(X_1,\ldots,X_n)\theta \in \mathcal{S}^+(p(X_1,\ldots,X_n),\{c\})$ and $p(X_1,\ldots,X_n)\theta \in \mathcal{A}^-(p(X_1,\ldots,X_n))$.*
The converse does not hold.

Theorem 5.12 *The atom $p(X_1,\ldots,X_n)\theta$ is p-uncovered if and only if $p(X_1,\ldots,X_n)\theta \in \mathcal{A}^+(p(X_1,\ldots,X_n))$ and $p(X_1,\ldots,X_n)\theta \notin \mathcal{S}^+(p(X_1,\ldots,X_n),P)$.*

The corresponding PROLOG meta-program is shown in Figure 3.

Note that now the search is driven by the elements in the negative and positive specification, obtained from the corresponding oracles. Both oracles nondeterministically instantiate their argument. An extensional implementation of the oracles requires

i. a unit clause of the form `panswer(A).` for any $A \in \mathcal{I}^+$,


```
append([A],B,B).
append([A|B],C,[A,D]) :- append(B,C,D).
```

Figure 4: A wrong acceptable program

ii. a unit clause of the form `nanswer(A).` for any $A \in \mathcal{I}^-$,

iii. the clause `panswer((A,G)) :- panswer(A),panswer(G).`, to get the intended (positive) answers for conjunctive goals.

Finally, we look at a small example, which shows that it is convenient to use finite subsets of the s-semantics, since their elements may still represent infinite sets of ground atoms.

Example 5.13 Consider the acceptable program P of figure 4, whose first clause is wrong. The partial specification is

$$\mathcal{I}^+ = \{\, append([],X,X),\ append([A],X,[A|X])\} \qquad \mathcal{I}^- = \{\, append([A],X,X)\}.$$

The diagnosis delivers the following result:

i. the clause `append([A],B,B).` is p-incorrect on $append([A],X,X)$.

ii. the atom $append([],X,X)$ is p-uncovered.

∎

6 Conclusions

Our first result is the extension of known diagnosis methods based on the detection of incorrect clauses and uncovered atoms to the case of the s-semantics. The good news is that absence of uncovered atoms implies completeness, for a large class of interesting programs (acceptable programs).

The second result is the definition of a top-down diagnoser, which has the following features: it uses one oracle only, it does not require to determine in advance the symptoms and is driven by the (finite) set of most general atomic goals, it is not subject to the incompleteness problem of Ferrand's diagnoser (since the s-semantics is not closed under instantiation).

Finally, we have introduced the diagnosis w.r.t. partial specifications, which leads to an effective diagnosis method, which is weaker than the general one in the case of correctness, yet can efficiently be implemented in both a top-down and in a bottom-up style.

All the results can naturally be extended to the more general framework of abstract diagnosis.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

"Optimal" Collecting Semantics for Analysis in a Hierarchy of Logic Program Semantics

Roberto Giacobazzi*

LIX, Laboratoire d'Informatique

École Polytechnique, 91128 Palaiseau cedex (France)

E-mail: giaco@lix.polytechnique.fr

Abstract

In this paper we define a framework of collecting semantics for analysis of logic programs. The idea is to use abstract interpretation to systematically derive, compose and compare semantics according to their expressive power. A hierarchy of collecting semantics is introduced, including well known semantics for logic programs and providing a formal basis to extend model theory to collecting and abstract semantics for analysis. We introduce a formal definition of *adequacy* for a semantics with respect to dataflow analysis, and a constructive characterization for the "best" collecting semantics for analysis.

1 Introduction

The definition of an appropriate concrete semantics, being able to model those program properties of interest, is a key point in abstract interpretation ([12]) and semantic-based dataflow analysis. As shown in [17] the choice of the operational semantics is usually the most appropriate one, as it is possible to derive more abstract semantics (e.g., the denotational semantics) by abstract interpretation. This leads to a *hierarchy of semantics* where well known semantics at different levels of abstraction are all derived by abstract interpretation from the operational one [14]. However, more abstract semantic bases can be suitable to avoid unnecessary details which are useless with respect to the program properties of interest. This is particularly important to simplify proofs of soundness in semantic-based static analysis. Of course, the best choice for a semantics should be a semantics which is not too abstract to hide too many details, but also not too concrete to introduce useless information (usually encoded by too complex semantic structures). A collecting semantics is somehow an intermediate step in abstraction between an often too concrete operational semantics and the standard semantics of the program (e.g., see the step-by-step abstraction in [29]). These semantics are usually derived by abstraction from an operational semantics of the language, or derived by a simple concretization process based on a powerset construction, collecting sets of denotations. Therefore, the relation between collecting semantics and the underlying more abstract standard semantics for the language becomes purely artificial and it is often meaningless. In logic programming for instance, it is often the case that collecting interpretations are derived by abstracting SLD resolution, without providing any corresponding model-theoretic interpretation, and the collecting semantics result to be often too far from the intended logical meaning of the program: its *Herbrand model* (e.g., see the operational frameworks in [8, 18], or the denotational semantics in [26]).

In this paper we introduce a new approach to collecting semantics design and analysis, and apply it to the case of logic programs. Collecting semantics are here characterized by maintaining the underlying structure of a standard semantics (later in the paper called *core semantics*), which is characterized by the so called "no junk" and "no confusion" conditions, providing a kind of *minimality* with respect to a given semantic property. Therefore, a collecting semantics is not merely a sound approximation of a more concrete semantic definition, but has to *include* a more

*This work has been partly supported by the EEC *Human Capital and Mobility* individual grant: "Semantic Definitions, Abstract Interpretation and Constraint Reasoning", N. ERB4001GT930817.