PROLE

Cooperation of the Finite Domain and Set Solvers in $\mathcal{TOY}$

S. Estévez-Martín[†], A. J. Fernández[‡], and F. Sáenz-Pérez[†1]

[†] *Dept. Sistemas Informáticos y Programación*, [‡] *Dept. Ingeniería del Software e Inteligencia Artificial*
Universidad Complutense de Madrid, Spain
[‡] *Dept. Lenguajes y Ciencias de la Computación*
Universidad de Málaga, Spain
s.estevez@fdi.ucm.es, afdez@lcc.uma.es, fernan@sip.ucm.es

Abstract

This paper describes the cooperation mechanism between two constraint domains, namely Finite Domain and Set, that has been implemented in the $\mathcal{TOY}$ system. The former one was already described in previous works, whereas the latter is a new addition to $\mathcal{TOY}$, which is a novelty presented here for the first time. This work introduces the Set solver, shows its components (data values, functions, and operators), describes a number of details about its implementation, and also explains the way in which the two mentioned domains cooperate. In addition, the paper provides a number of examples that highlight the functionalities of the Set solver and also illustrate the feasibility of our cooperation proposal.

Keywords: Constraints, Solver Cooperation, Finite Domains, Sets, Functional Logic Programming

1 Introduction

Nowadays, applications arise in highly heterogeneous scenarios requiring the integration of several software technologies. Satisfaction and optimization applications are revealed as critical examples for which different data structures and types are needed to be combined. On the one hand, there exists the need to integrate these different data domains, whilst, on the other, specific solving techniques are demanded to compute in each application domain. High-level programming systems help to develop such applications by abstractions, as the use of constraints. Examples of such systems lay in the declarative programming paradigm.

Here, we focus on the constraint functional logic programming (CFLP) system $\mathcal{TOY}$ [1], a state-of-the-art representative of this paradigm, which embodies several constraint domains and allows their integration and cooperation. It implements the cooperation mechanisms presented in [4]. In this paper, we propose the integration

[1] First and third authors partially supported by projects TIN2005-09207-C03-03, TIN2008-06622-C03-01, S-0505/TIC/0407 and UCM-BSCH-GR58/08-910502. Second author partially supported by Spanish MICINN under contract TIN2008-05941 (NEMESIS project)

of the new constraint domain *Set* which implements data types, functions and operators on polymorphic sets. This constraint domain is based on interval lattices [6,7] and, in addition, a bridge allowing the communication and cooperation between the finite domain and set constraint domains is proposed. An example supporting the applicability of this integration is presented.

2 Related Work

Most of the existing constraint solvers only provide support for solving on specified domains: this implies that the constraints are usually restricted to just values in the given computation domain. However, in practice, many problems require heterogeneous constraints expressed using more than one domain and, thus, problem representations have to be artificially adapted to fit the particular choice of domain and solver. A solution to this problem has been found in the concept of *solver cooperation*, an issue that has attracted the attention of many researchers in the constraint programming (CP) community. Here we mention [2,10,12,15] as a limited selection of references illustrating various approaches to the problem.

In this paper we consider the cooperation between the finite and set domains. The *finite domain* is very well-known in CP due to its practicability [11] whereas *Set* is another important domain employed traditionally in the constraint logic programming (CLP) setting because sets enable the modelling of problems formulated on combinatorial solving and on natural language processing. In the literature one can find a number of proposals to integrate sets in constraint solvers [3,9,13,14,16].

3 An Overview of $\mathcal{TOY}(\mathcal{S})$

The set constraint solver $\mathcal{S}$ ($solve^{\mathcal{S}}$) used in the $\mathcal{TOY}$ system has been implemented as a particular instance of the schema described in [6,7]. This schema, based on interval lattices, is a generic framework for both interval constraint satisfaction and interval solver cooperation, and defines a constraint solving proposal for CLP(Interval($\mathcal{X}$)), i.e., CLP on intervals defined on any ***computation domain*** $\mathcal{X}$ with lattice structure. Here, $\mathcal{X}$ is the lattice $\mathcal{S}_\Delta$, i.e., the domain of sets defined on a *basic domain* Δ (e.g., integers or characters); our $\mathcal{S}$ solver is initially developed as a tool for CLP(Interval($\mathcal{S}_\Delta$)), that is to say, a solver for CLP over intervals of sets defined on Δ. This solver is further integrated into the CFLP system $\mathcal{TOY}$.

The computation domain $\mathcal{S}_\Delta$ is a lattice where $\subseteq$, $\cup$ and $\cap$ define respectively a partial order on the computation domain, the *join* or least upper bound, and the *meet* or greatest upper bound. The bottom element of this lattice is the empty set (denoted as $\{\}$) and the top element is the set containing all the elements of the basic domain Δ (denoted as 'top').

This section is devoted to describe the $\mathcal{TOY}(\mathcal{S})$ solver. We show the main components of our solver, illustrate how constraint solving is defined, and describe briefly its implementation. A number of simple examples that highlight the functionalities of this solver are also provided in order to help the reader to understand the set interval constraint solving process integrated into the $\mathcal{TOY}$ system.

3.1 *Constraint Solving in $\mathcal{TOY}(\mathcal{S})$ Solver*

Our $\mathcal{TOY}(\mathcal{S})$ solver keeps many similarities with *Conjunto* [9]. However, there are also many differences with respect to *Conjunto* that were already highlighted in [6,7]. In the following, we provide a brief description of our solver.

$\mathcal{TOY}(\mathcal{S})$ solver uses an instance over integers of the solver described in [6,7], allowing to solve set interval constraints, where a *set interval constraint* is an expression of the form:

$$S_{int} \in [Min, Max]$$

and Min and Max are sets of elements belonging to int, the set of integers. The constraint is consistent iff $Min \subseteq Max$ and then the meaning is basically that S_{int} can have (set) values in the *set interval* $[Min, Max]$. In other words, S_{int} can be any concrete set s such that $Min \subseteq s \subseteq Max$. Otherwise, that is to say $Min \nsubseteq Max$, the constraint is inconsistent since there are no values to which variable S_{int} can be assigned.

In the rest of the paper we will omit the subindex int in S_{int} as this should be clear from the context. Also, set interval constraints will be named as $\mathcal{S}$ constraints.

Example 3.1 Let S be an integer set variable. Then, a constraint such as $S \in [\{\}, \{1,2\}]$ means that S might be one of the following set values: the empty set $\{\}$, the set $\{1\}$, the set $\{2\}$, or the set $\{1,2\}$. Also, a constraint $S \in [\{3\}, \{3,4\}]$ means that S might be one of the following set values: the set $\{3\}$ or the set $\{3,4\}$.

$\mathcal{TOY}(\mathcal{S})$ propagates set interval constraints by imposing bound consistency. It is expected then that constraint propagation will lead to further ***constraint narrowing*** that consists basically of the reduction of the set intervals (i.e., the domains) associated to each set variable. Pruning the domains associated to the set variables (i.e., those set values that the variable can be assigned to) is done by refining the bounds of their corresponding associated set intervals. This idea is not new and was already proposed in [9] although in our case is a direct consequence of implementing the solver as an instance of the CLP(Interval($\mathcal{X}$)), for $\mathcal{X} = \mathcal{S}_{int}$, as already mentioned. In a very simplified form, the domain reduction consists of both increasing the lower bound and decreasing the upper bound of set intervals associated to set variables by, respectively, adding and removing elements in the bounds.

Example 3.2 Let $S \in [\{\}, \{1,2\}]$ be an interval constraint. Then, for instance, this constraint can be narrowed by removing the value 2 from the upper bound: in this case the constraint is reduced to $S \in [\{\}, \{1\}]$ and thus S only might have two set values, namely $\{\}$ or the singleton $\{1\}$.

3.2 *$\mathcal{TOY}(\mathcal{S})$ Programs*

$\mathcal{TOY}(\mathcal{S})$ programs are $\mathcal{TOY}$ programs where $\mathcal{S}$ constraints are defined as functions which are solved by our $\mathcal{S}$ solver connected to $\mathcal{TOY}$. Programs in $\mathcal{TOY}(\mathcal{S})$, as in $\mathcal{TOY}$, can include definitions of types, operators, lazy functions in Haskell style, as well as definitions of predicates in Prolog style. The syntax is mostly borrowed from Haskell with the remarkable exception that variables begin with upper-case whereas constructor symbols use lower-case, as function symbols do.

Each function f has an associated principal type of the form $\tau_1 \to \cdots \to \tau_n \to \tau$. As usual in functional programming, types are inferred and, optionally, can be declared in the program. Defining rules for a function f have the basic form $f\ t_1 ... t_n = r \Leftarrow C$. Informally, the intended meaning of a program rule is that a call to f can be narrowed to r whenever the actual parameters match the patterns t_i, and the conditions in C are satisfied. An n-ary predicate is viewed as a particular kind of function, with type $\tau_1 \to \ldots \to \tau_n \to bool$. As a syntactic facility, we can use *clauses* as a shorthand for defining rules whose right-hand side is true. This allows one to write Prolog-like predicate definitions: each clause $p\ t_1 \ldots t_n :- C_1, \ldots, C_k$ abbreviates a defining rule of the form $p\ t_1 \ldots t_n = true \Leftarrow C_1, ..., C_k$.

3.3 Set Constraints in $\mathcal{TOY}(\mathcal{S})$

The available set constraints and functions we define is as follows, where `int`, `bool`, and `setOfInt` denote the type for integers, Booleans, and set of integers, resp. This last type can be read as the datatype declaration `data setOfInt = set [int]`.

- `domainSet` $[S_1, \ldots, S_l]$ `(set` $[X_1, \ldots, X_k]$`) (set` $[Y_1, \ldots, Y_l]$`)` (with type `[setOfInt] -> setOfInt -> setOfInt -> bool`) imposes the following conjunction of set interval constraints:

$$\bigwedge_{1 \leq i \leq l} S_i \in [\{X_1, \ldots, X_k\}, \{Y_1, \ldots, Y_l\}].$$

- `subSet` S_1 S_2 (with type `setOfInt -> setOfInt -> bool`) imposes that each element of the set S_1 belongs to the set S_2, reducing the domain of S_1 if necessary. Analogous for `superSet`.
- `intersectSet` S_1 S_2 (with type `setOfInt -> setOfInt -> setOfInt`) returns the set intersection of S_1 and S_2.
- `unionSet` S_1 S_2 (with type `setOfInt -> setOfInt -> setOfInt`) returns the set union of S_1 and S_2.
- `subtractSet` S_1 S_2 (with type `setOfInt -> setOfInt -> setOfInt`) returns a set with the elements in S_1 that do not occur in S_2.
- `removeFromSet A` S (with type `int -> setOfInt -> setOfInt`) returns a set with the elements in S but the element `A`.
- `cardinalSet` S (with type `setOfInt -> int`) returns the cardinality of S.
- `labelingSet` $[S_1, \ldots, S_n]$ (with type `[setOfInt] -> bool`) provides a set variable enumeration strategy and returns `true` if an assignment for the sets $S_1, \ldots, S_n$ can be found, such that all of the constraints in the set constraint store are satisfied.

Regarding `labelingSet/1`, we note that this function is defined in order to provide a complete $\mathcal{TOY}(\mathcal{S})$ solver as it is well-known that, in general, propagation algorithms are not enough to solve a constraint satisfaction problem (CSP) and, as a consequence, it is very frequent that, when no more constraint propagation is possible, an additional labeling strategy is employed to solve the CSP. This process is very usual in association with solvers over finite and discrete domains.

3.4 Inference Rules

In this subsection we show (cf. Table 1, where S_i, $1 \leq i \leq 3$, denotes a set variable, F a finite domain variable, $\#s$ the cardinality of a set s, and, as usual, $A \setminus B$ the set difference) a number of inference rules that guide the process of set interval constraint solving in $\mathcal{TOY}(\mathcal{S})$. Constraint solving is executed as a combination of constraint propagation, constraint narrowing, and variable enumeration. Both constraint narrowing and enumeration of set variables are a direct consequence of applying their corresponding inference rules; constraint propagation is produced via the application of the rest of rules.

Rule Name	Inference rules
`Domain`	$\dfrac{S \in [s,s]}{S == s}$
`Inconsistency`	$\dfrac{S \in [a,b],\ a \not\subseteq b}{\text{fail}}$
`Subset`	$\dfrac{S_1 \in [a,b],\ S_2 \in [c,d],\ S_1 \subseteq S_2}{S_1 \in [a,d],\ S_2 \in [a,d]}$
`Superset`	$\dfrac{S_1 \supseteq S_2}{S_2 \subseteq S_1}$
`Union`	$\dfrac{S_1 \in [a,b],\ S_2 \in [c,d],\ S_3 \in [e,f],\ S_1 \cup S_2 == S_3}{S_1 \in [e \setminus d, f],\ S_2 \in [e \setminus b, f],\ S_3 \in [a \cup c, b \cup d]}$
`Intersection`	$\dfrac{S_1 \in [a,b],\ S_2 \in [c,d],\ S_3 \in [e,f],\ S_1 \cap S_2 == S_3}{S_1 \in [e \cap c, top],\ S_2 \in [e \cap a, top],\ S_3 \in [a \cap c, b \cap d]}$
`Subtract`	$\dfrac{S_1 \in [a,b],\ S_2 \in [c,d],\ S_3 \in [e,f],\ S_1 \setminus S_2 == S_3}{S_1 \in [\{\}, d \cup f],\ S_2 \in [\{\}, top],\ S_3 \in [a \setminus d, b \setminus c]}$
`CardinalSet`	$\dfrac{S == s,\ \texttt{cardinalSet}\ S == F}{F == \#s}$ $\dfrac{S \in [a,b],\ \texttt{cardinalSet}\ S == F}{F \in [\#a, \#b]}$
`Constraint Narrowing (CN)`	$\dfrac{S \in [a,b],\ S \in [c,d]}{S \in [a \cup c, b \cap d]}$
`Set Labeling`	$\dfrac{S == s,\ \texttt{labelingSet}\ [S]}{\text{true}}$ $\dfrac{S \in [a,b],\ \texttt{labelingSet}\ [S]}{S \in [a \cup \{x\}, b]\ \texttt{or}\ S \in [a, b \setminus \{x\}],\ \text{for some } x \in b \text{ and } x \notin a}$ $\dfrac{\texttt{labelingSet}\ [S \mid S_s]}{\texttt{labelingSet}\ [S],\ \texttt{labelingSet}\ S_s}$

Table 1
Some Inference Rules for the Set Interval Constraint Solver

Example 3.3 In the following we illustrate a process of constraint solving (without labeling) where inference rules have been applied in an arbitrary order. $\Rightarrow^R$ denotes the rule R applied in each step, and the selected $\mathcal{TOY}(\mathcal{S})$ constraints for this rule in the inference process are underlined. Note that, opposed to usual deductions, we show some intermediate steps which are performed by the constraint solver and not described by the inference rules. In particular, by applying the Subset rule, only one deduction step should appear. Here, we add conclusions without removing hypotheses but when pruning domain by merging them via the CN rule.

$\underline{S_1 \in [\{1\},\{1,2,3\}]}$, $\underline{S_2 \in [\{\},\{1,2,3,4\}]}$, $\underline{S_1 \subseteq S_2} \Rightarrow^{\text{Subset}}$

$\underline{S_1 \in [\{1\},\{1,2,3\}]}$, $S_2 \in [\{\},\{1,2,3,4\}]$, $S_1 \subseteq S_2$, $\underline{S_1 \in [\{1\},\{1,2,3,4\}]}$, $S_2 \in [\{1\},\{1,2,3,4\}] \Rightarrow^{\text{CN}}$

$S_1 \in [\{1\},\{1,2,3\}]$, $\underline{S_2 \in [\{\},\{1,2,3,4\}]}$, $S_1 \subseteq S_2$, $\underline{S_2 \in [\{1\},\{1,2,3,4\}]} \Rightarrow^{\text{CN}}$

$S_1 \in [\{1\},\{1,2,3\}]$, $S_2 \in [\{1\},\{1,2,3,4\}]$, $S_1 \subseteq S_2$

Note that some rules might have been combined to obtain further domain reduction. For instance, CN and Subset rules can be combined as follows:

$$\frac{S_1 \in [a,b],\ S_2 \in [c,d],\ S_1 \subseteq S_2}{S_1 \in [a, b \cap d],\ S_2 \in [c \cup a, d]}$$

$\mathcal{TOY}(\mathcal{S})$ has been implemented in SICStus Prolog via Constraint Handling Rules (CHRs) [8]: these rules are a proposal to allow more flexibility and application-oriented customization of constraint systems. In this paper we will neither discuss optimizations nor further implementation issues as this is a topic of further work.

4 Cooperation between the Domains $\mathcal{S}$ and $\mathcal{FD}$

In this section, we explain the cooperation among the pure constraint domains $\mathcal{S}$, just described, and $\mathcal{FD}$, which supplies arithmetic and finite domain constraints over integers (see [1,5]). Table 2 recalls some primitive functions.

Our cooperative computation model keeps different stores for constraints corresponding to different domains. In addition, there is a special store, called *mediatorial store* ($\mathcal{M}_{\mathcal{FS}}$) where the communication constraints between the finite domain and the set domain are placed. Communication constraints (`F #-- S`, following the type in Table 2) are called *bridges* and can be used to project constraints involving the variable S into constraints involving the variable F, as we will see in Section 4.1.

4.1 Projections from $\mathcal{S}$ to $\mathcal{FD}$

Projection takes place whenever a constraint is submitted to the $\mathcal{S}$ solver. At that moment, projection rules relying on the available cooperation constraints (`#--`) are used for building mate constraints which are submitted to the $\mathcal{FD}$ solver. Table 3 shows the available opportunities for projections, assuming there exist bridges `Fi #-- Si`.

The second column of this table shows new bridges calculated in the computing process, where the symbol - means that no bridges are created. The third column shows the $\mathcal{FD}$ constraints which are projected, where they correspond to the constraints listed in the $\mathcal{TOY}$ user manual [1], but `setcomplement F1 F2` (constrains

Domain	Primitive Functions

$\mathcal{FD}$	COMPARISON CONSTRAINTS
	`#=, #>, #<, #>=, #<= :: int → int → bool`
	ARITHMETICAL OPERATORS
	`#*, #/, #+, #- :: int → int → int`
	COMBINATORIAL CONSTRAINT
	`all_different :: [int] → bool`
	MEMBERSHIP CONSTRAINTS
	`domain :: [int] → int → int → bool`
	`subset, setcomplement :: int → int → bool`
	`intersect, union :: int → int → int`
	PROPOSITIONAL CONSTRAINT
	`#<=> :: bool → bool → bool`
	ENUMERATION FUNCTION
	`labeling :: [labelType] → [int] → bool`
	REFLECTION FUNCTIONS
	`fd_min, fd_max, fd_size, fd_degree :: int → int`
	`fd_dom :: int → fdrange`

$\mathcal{M}_{\mathcal{FS}}$	`#-- :: int → setOfInt → bool`

Table 2
$\mathcal{FD}$ and $\mathcal{M}_{\mathcal{FS}}$ Constraints, Operators and Functions.

Constraint	*Bridges*	*Projections*
`domainSet [S1,...,Sn] [X1,...,Xj] [Y1,...,Yk]`		`domain [F1,...,Fn] Y1 Yk`
`subSet S1 S2` (resp `superSet`)	-	`subset F1 F2`
`intersectSet S1 S2 S3`	`F3 #-- S3`	`intersect F1 F2 == F3`
`unionSet S1 S2 S3`	`F3 #-- S3`	`union F1 F2 == F3`
`subtractSet S1 S2 S3`	`F3 #-- S3`	`domain [F3] (fd_min F1) (fd_max F1), setcomplement F3 F2`
`removeFromSet A S1 S2`	`F2 #-- S2`	`domain [F2] (fd_min F1) (fd_max F1), F2 /= A`
`cardSet S1 N`	-	`card F1 == N`

Table 3
Computing Bridges and Projections from $\mathcal{S}$ to $\mathcal{FD}$.

the domain of F1 to be the complemented domain of F2), `union F1 F2 == F3` (constrains the domain of F3 to the union of the domains F1 and F2), and `card S1 == N` (constrains the cardinality of S1 to N), which are new $\mathcal{FD}$ additions unreported up to the next software release.

For example, the $\mathcal{TOY}$ goal

```
F1 #-- S1, F2 #-- S2,
domainSet [S1] (set [10]) (set [10,20]),
domainSet [S2] (set [10]) (set [10,20,30,40]),
intersectSet S1 S2 S3
```

creates a new bridge `F3 #-- S3` with a new variable F3. The projection of `domainSet` constraints cause that the variables F1 and F2 may take respectively the feasible values `10,20`, and `10,20,30,40`. Also, the constraint `intersectSet` prunes the domain of F3 to the feasible values 10,20.

Now, if the subSet S2 S1 constraint is added, then the domain of the variable S2 is reduced to {10,20}, and the corresponding projection reduces the domain of the variable F2 to 10,20.

4.2 A Cooperation Problem

We consider a program written in $\mathcal{TOY}$ for solving the problem of scheduling tasks that require resources to complete, and have to fulfil precedence constraints. Figure 1 shows a precedence graph for six tasks which are labelled as tX^{Y}_{mZ}, where X stands for the identifier of a task t, Y for its time to complete (duration), and Z for the identifier of a machine m (a resource needed for performing the task tX).

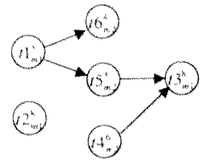

Fig. 1 Precedence Graph.

The following program models this scheduling problem.

```
(1) durationList :: [int]
    durationList = [3,8,8,6,3,4]

(2) listFrom1To :: int -> [int]
    listFrom1To X = take X (iterate (+ 1) 1)

(3) scheduling :: [setOfInt] -> [int] -> bool
    scheduling TasksSet TasksFD = true <==

(4)     TasksSet == [T1S, T2S, T3S, T4S, T5S, T6S],
(5)     TasksFD == [T1FD, T2FD, T3FD, T4FD, T5FD, T6FD],
(6)     foldl and true (zipWith (#--) TasksFD TasksSet),
(7)     sum durationList (#=) Time,
(8)     domainSet TasksSet (set []) (set (listFrom1To Time)),
(9)     map cardinalSet TasksSet == durationList,

(10)    %precedences
        fd_max T1FD #< fd_min T6FD,
        fd_max T1FD #< fd_min T5FD,
        fd_max T5FD #< fd_min T3FD,
        fd_max T4FD #< fd_min T3FD,

(11)    % machine m1
        intersectSet T1S T2S (set []),
        intersectSet T1S T3S (set []),
```



```
        intersectSet T2S T3S (set []),

(12)    %machine m2
        intersectSet T4S T5S (set []),
        intersectSet T5S T6S (set []),
        intersectSet T4S T6S (set [])
```

A task is modelled as a variable of type setOfInt. The time of execution of all tasks is, at most, the sum of the durations of all the tasks, that is, variable Time in line (7). Therefore, the time of executing every task can be placed in the time interval defined from 1 to Time (line (8)). Note that the empty list [] means that it is not possible to accomplish this task, and $[t_1, t_2, \ldots t_n]$ means that the tasks will be accomplished in the days $t_1, t_2, \ldots t_n$, where it is not necessary that they are consecutive days. The duration of a task corresponds to the cardinal of its set (line (9)).

The main function scheduling takes two arguments: the list of tasks to be scheduled as a list of sets, and the same list of tasks as a list of integer variables. Every set variable is related to a finite domain variable by the cooperation constraint #-- (line (6)). We use these variables in order to project set constraints into finite domain constraints when possible (cf. Subsection 4.1) and allow the cooperation of both solvers.

The tasks can use certain machines. If two tasks need to use the same machine then they cannot execute at the same time. Therefore, the intersection of the sets that represent them is empty (blocks (11) and (12)).

The precedences among tasks are represented by finite domain relational operators since this representation is simpler (block (10)).

Figure 2 shows a possible solution for the problem of scheduling tasks.

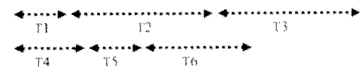

Fig. 2 A Possible Solution for the Problem of Scheduling Tasks

Projection is necessary to solve this problem, due to the fact that some constraints work in the $\mathcal{S}$ domain and others in the $\mathcal{FD}$ domain. The projection of the set constraints creates new finite domain constraints which are needed to solve the problem. In particular, the projection of domainSet prunes the domain of the finite domain variables to values included in 1 and Time. Next, cardinalSet, because of its projection, limits the number of elements in the domain of the mate finite domain variable to the number of days (in the list durationList). Finally, the intersectSet constraints project the finite domain constraints, therefore avoiding overlappings.

5 Conclusions and Future Work

We have presented a new domain $\mathcal{S}$ (i.e., the set of integers domain) integrated currently in the $\mathcal{TOY}$ system which can cooperate with the existing domain $\mathcal{FD}$.

The cooperation is allowed via a new bridge relating $\mathcal{S}$ and $\mathcal{FD}$ variables. Several constraint functions have been proposed and the $\mathcal{S}$ solver has been implemented using CHRs. We have shown an example of a scheduling problem with a more natural representation, which is in addition solved thanks to the cooperation of the solvers of both domains.

This work can be extended in a number of ways. For instance, we think to renew the current implementation of the set solver with a wider set primitive toolbox: this extension demands obviously the construction of additional bridges relating the set and finite domains. Also, we can study how to redefine the propagation mechanism of the set solver to produce stronger (bound) consistency. In addition, projection from $\mathcal{FD}$ to $\mathcal{S}$ will be considered as well as alternative models for the $\mathcal{FD} - \mathcal{S}$ bridges. Moreover, we are working on new examples to motivate the feasibility of our cooperative model.

References

[1] Arenas, P., S. Estévez, A. Fernández, A. Gil, F. López-Fraguas, M. Rodríguez-Artalejo and F. Sáenz-Pérez, *TOY. A Multiparadigm Declarative Language. Version 2.3.1*, (2007), R. Caballero and J. Sánchez (Eds.), Available at `http://toy.sourceforge.net`.

[2] Baader, F. and K. Schulz, *On the combination of symbolic constraints, solution domains and constraints solvers*, in: *CP'95*, LNCS **976** (1995), pp. 380–397.

[3] Dovier, A., E. Omodeo, E. Pontelli and G. Rossi, *{log}: a language for programming in logic with finite sets*, Journal of Logic Programming **28** (1996), pp. 1–44.

[4] Estévez-Martín, S., A. J. Fernández, T. Hortalá-González, M. Rodríguez-Artalejo, F. Sáenz-Pérez and R. del Vado Vírseda, *On the Cooperation of the Constraint Domains H, R and FD in CFLP*, Theory and Practice of Logic Programming (2009), accepted for publication. `http://arxiv.org/abs/0904.2136`.

[5] Fernández, A., M. Hortalá-González and F. Sáenz-Pérez et al., *TOY(FD): User's manual*, `http://toy.sourceforge.net`

[6] Fernández, A. J. and P. M. Hill, *An interval constraint system for lattice domains.*, ACM Transactions on Programming Languages and Systems **26** (2004), pp. 1–46.

[7] Fernández, A. J. and P. M. Hill, *An interval constraint branching scheme for lattice domains*, J. UCS **12** (2006), pp. 1466–1499.

[8] Frühwirth, T., *Theory and practice of constraint handling rules*, The Journal of Logic Programming **37** (1998), pp. 95–138.

[9] Gervet, C., *Interval propagation to reason about sets: definition and implementation of a practical language*, Constraints **1** (1997), pp. 191–244.

[10] Granvilliers, L., E. Monfroy and F. Benhamou, *Cooperative solvers in constraint programming: a short introduction*, ALP Newsletter **14** (2001).

[11] Henz, M. and T. Müller, *An overview of finite domain constraint programming*, in: *5th Conference of the Association of Asia Pacific Operational Research Societies*, Singapore, 2000.

[12] Hofstedt, P. and P. Pepper, *Integration of declarative and constraint programming*, Theory and Practice of Logic Programming **7** (2007), pp. 93–121.

[13] Kozen, D., *Set constraints and logic programming*, Information and Computation **142** (1998), pp. 2–25.

[14] Legeard, B. and E. Legros, *Short overview of the CLPS system*, in: J. Maluszynski and M. Wirsing, editors, *3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP'97)*, number 528 in LNCS (1991), pp. 431–433.

[15] Monfroy, E. and C. Castro, *A component language for hybrid solver cooperations*, in: *ADVIS'04*, LNCS **3261** (2004), pp. 192–202.

[16] Walinsky, C., *CLP(Σ*): constraint logic programming with regular sets*, in: G. Levi and M. Martelli, editors, *6th International Conference on Logic Programming (ICLP'89)* (1989), pp. 181–196.

Advances in type systems for Functional-Logic Programming
(*Work in progress*)

Francisco López-Fraguas[1], Enrique Martin-Martin[2] and Juan Rodríguez-Hortalá[3]

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Madrid, Spain

Abstract

Type systems are widely used in programming languages as a powerful tool providing safety to programs, and forcing the programmers to write code in a clearer way. Functional-logic languages have inherited Damas & Milner type system from their functional part due to its simplicity and popularity. However, functional-logic languages have some problematic features not taken under consideration by standard systems. In particular, it is known that the use of *opaque* HO patterns in left-hand sides of program rules may produce undesirable effects from the point of view of types. We re-examine the problem, and propose a Damas & Milner-like type system where certain uses of HO patterns (even opaque) are permitted while preserving type safety, as proved by a subject reduction result that uses a *HO-let-rewriting*, a recently proposed operational semantics for HO functional logic programs.

Keywords: Functional logic programming, Type systems, Opaque patterns

1 Introduction

Type systems for programming languages are an active area of research [16], no matters which paradigm one considers. In the case of functional programming, most type systems have arisen as extensions of Damas & Milner's [4], for its remarkable simplicity and good properties (decidability, existence of principal types, possibility of type inference). Functional logic languages [11,8,7], in their practical side, have inherited more or less directly Damas & Milner's types. In principle, most of the type extensions proposed for functional programming could be also incorporated to functional logic languages (this has been done, for instance, for type classes in

* This work has been partially supported by the Spanish projects TIN2005-09207-C03-03, TIN2008-06622-C03-01, S-0505/TIC/0407 and UCM-BSCH-GR58/08-910502

[1] Email: `fraguas@sip.ucm.es`

[2] Email: `emartinm@fdi.ucm.es`

[3] Email: `juanrh@fdi.ucm.es`

IX Jornadas sobre Programación y Lenguajes
I Taller de Programación Funcional
P. Lucio, G. Moreno y R. Peña, Eds.
San Sebastián, 8-11 de septiembre de 2009

Programación y Lenguajes

IX Jornadas sobre Programación y Lenguajes, PROLE'09
I Taller de Programación Funcional, TPF'09

San Sebastián, España

del 8 al 11 de Septiembre de 2009

Editores:

Paqui Lucio, Ginés Moreno y Ricardo Peña

:dita:
'aqui Lucio Carrasco
;inés Moreno Valverde
{icardo Peña Marí

'ilmación e impresión:
;ráficas Michelena

)epósito Legal:
;S-990-2009

SBN:
'78-84-692-4600-9

Comité de Programa de PROLE-2009

Presidente Comité: Ginés Moreno (U. de Castilla-La Mancha)

Elvira Albert (U. Complutense de Madrid)
Jesús Almendros (U. de Almería)
María Alpuente (U. Politécnica de Valencia)
Gilles Barthe (IMDEA)
Miquel Bertran (U. Ramón Llull)
Santiago Escobar (U. Politécnica de Valencia)
Antonio Fernández (U. de Málaga)
Victor Gulías (U. de A Coruña)
Paqui Lucio (U. del País Vasco)
Narciso Martí-Oliet (U. Complutense de Madrid)
Susana Muñoz (U. Politécnica de Madrid)
Marisa Navarro (U. del País Vasco)
Manuel Núñez (U. Complutense de Madrid)
Fernando Orejas (U. Politécnica de Catalunya)
Yolanda Ortega (U. Complutense de Madrid)
Francisco Ortín (U. de Oviedo)
Ernesto Pimentel (U. de Málaga)
Germán Puebla (U. Politécnica de Madrid)
Enric Rodríguez (U. Politécnica de Catalunya)
Jaime Sánchez (U. Complutense de Madrid)
Germán Vidal (U. Politécnica de Valencia)

Comité de Programa de TPF-2009

Presidente Comité: Ricardo Peña (U. Complutense de Madrid)

Victor Gulías (U. de A Coruña)
Francisco Gutiérrez (U. de Málaga)
Salvador Lucas (U. Politécnica de Valencia)
Pablo Nogueira (U. Politécnica de Madrid)
Julio Rubio (U. de la Rioja)
Alberto Verdejo (U. Complutense de Madrid)
Mateu Villaret (U. de Girona)

Comité Organizador de JISBD/PROLE-2009

Comité Ejecutivo de PROLE-2009

Revisores adicionales de PROLE/TPF-2009

Índice

Prólogo

Las Jornadas sobre PROgramación y LEnguajes (PROLE) se vienen consolidando como un marco propicio de reunión, debate y divulgación para los grupos españoles que investigan en temas relacionados con la programación y los lenguajes de programación. La investigación en este campo está en continuo desarrollo y comprende todo el estudio de conceptos, métodos, técnicas, fundamentos y aplicaciones relativos a la tarea de programar y a los lenguajes que se utilizan en ella. El evento, de carácter anual, pretende fomentar tanto el intercambio de experiencias y resultados, como la comunicación y cooperación entre los grupos de investigadores españoles que trabajan en el área de programación y lenguajes, manteniendo un año más la enriquecedora trayectoria de las ocho ediciones previas celebradas en Almagro (2001), El Escorial (2002), Alicante (2003), Málaga (2004), Granada (2005), Sitges (2006), Zaragoza (2007) y Gijón (2008).

En esta ocasión, la IX edición de las Jornadas (PROLE'09) va precedida por primera vez en su historia del I Taller sobre Programación Funcional (TPF'09). Ambos eventos se celebran entre el 8 y el 11 de septiembre de 2009, dentro de la XXVIII edición de los Cursos de Verano de San Sebastián. Como en ocasiones previas, la organización de esta conferencia se realiza en paralelo con las Jornadas de Ingeniería del Software y Bases de Datos (JISBD'09), compartiendo conferencias invitadas, actos sociales, publicidad, etc. Para información más detallada puede consultarse `http://www.mondragon.edu/prole2009/`. La organización conjunta de ambos eventos ha sido auspiciada por la Sociedad de Ingeniería del Software y Tecnologías de Desarrollo de Software (SISTEDES). Agradecemos desde aquí el soporte, la infraestructura y el apoyo prestado por todos los agentes arriba mencionados.

En el ámbito de PROLE'09 se han seleccionado este año un total de 31 trabajos, que cubren tanto aspectos teóricos como prácticos relativos a la especificación, diseño, implementación, análisis y verificación de programas y lenguajes de programación, además de herramientas tangibles y sistemas software que incrementan el carácter pragmático del área. Por su parte, el Taller de Programación Funcional TPF'09 que precede a PROLE'09, inicia su recorrido como una actividad independiente y complementaria a PROLE, con un comité de programa propio que ha seleccionado 5 trabajos recogidos también en estas actas, y que se centran en aspectos relacionados con lenguajes de programación con una fuerte componente funcional (en esta edición los trabajos se refieren a los lenguages Haskell, Lisp y Maude), incluyendo herramientas y experiencias docentes y de investigación en torno a este tipo de lenguajes. Como parte del Taller, se celebra también en esta ocasión una mesa redonda acerca de la inclusión de la programación funcional y lógica en los nuevos planes de Grado que empezarán a impartirse en 2009.

Este volumen recopila por tanto un total de 36 trabajos que fueron rigurosamente revisados cada uno de ellos por 3 miembros de ambos comités de programa y/o revisores adicionales, a los cuales es necesario agradecer su inestimable ayuda y reconocer su gran profesionalidad. También en consonancia con este agradecimiento, es justo felicitar a los autores por la calidad de sus trabajos y su contribución a que esta edición sea la de mayor participación en la evolución histórica de PROLE, lo que garantiza la buena salud del evento.

Por otro lado, además de las tres conferencias invitadas que compartimos con la planificación de JISBD'09, el programa de PROLE'09 cuenta este año con una excelente conferencia específica (un resumen de la misma se incluye en este volumen) que, bajo el título de "Understanding program verification", será impartida por K. Rustan M. Leino, de Microsoft Research, USA, a quien agradecemos el haber aceptado tan amablemente nuestra invitación.

Finalmente, queremos agradecer la confianza que han depositado en nosotros, para conducir la presente edición de estas jornadas, a todos los miembros del comité ejecutivo de PROLE, y esperamos no haberles defraudado. En el desempeño de esta tarea, ha sido determinante la ayuda y experiencia prestada por Jesús Almendros, quien presidió la anterior edición de PROLE en Gijón, y a quien aprovechamos para dar mil gracias desde aquí.

Septiembre de 2009

Paqui Lucio
Ginés Moreno
Ricardo Peña

PATROCINADORES

MONDRAGON UNIBERTSITATEA

Sistedes

Universidad del País Vasco / Euskal Herriko Unibertsitatea

EUSKO JAURLARITZA / GOBIERNO VASCO

GOBIERNO DE ESPAÑA

MINISTERIO DE CIENCIA E INNOVACIÓN

MONDRAGON

INTERSYSTEMS

ati Asociación de Técnicos de Informática

novática UPGRADE

CAJA LABORAL EUSKADIKO KUTXA

ULMA