Adding Partial Functions to Constraint Logic Programming with Sets

Maximiliano Cristiá[1] and Gianfranco Rossi[2]

[1] CIFASIS and UNR, Rosario, Argentina
cristia@cifasis-conicet.gov.ar
[2] Università degli studi di Parma, Parma, Italy
gianfranco.rossi@unipr.it

Abstract. Partial functions are common abstractions in formal specification notations such as Z, B and Alloy. Conversely, executable programming languages usually provide little or no support for them. In this paper we propose to add partial functions as a primitive feature to a Constraint Logic Programming (CLP) language, namely $\{log\}$. Although partial functions could be programmed on top of $\{log\}$, providing them as first-class citizens adds valuable flexibility and generality to the form of set-theoretic formulas that the language can safely deal with. In particular, the paper shows how the $\{log\}$ constraint solver is naturally extended in order to accommodate for the new primitive constraints dealing with partial functions.

1 Introduction

Given any two sets, X and Y, a *binary relation between X and Y* is any subset of the power set of $X \times Y$, $\mathbb{P}(X \times Y)$. Partial functions are just a particular kind of binary relations, in which ordered pairs are restricted to verify the classical notion of function—i.e. that each element in the domain is mapped to at most one element in the range—, although it may be undefined for every element in the domain—i.e. it is partial. Binary relations are in turn just sets of ordered pairs. Then, all relational operators (such as dom, ran, $\fatsemi$, etc.) can be applied to partial functions and all set operators can be applied to both of them. Conversely, and this feature distinguishes partial functions from binary relations, if x is an element in the domain of partial function f then $f(x)$ is defined as the element, y, in the range of f such that $(x, y) \in f$.

Partial functions are common in formal specification notations, such as Z [9], B [1] and Alloy [6], which are mainly used to specify state-based systems. Many concepts or features of these systems are best represented as partial functions, not as total functions.

In previous work [5] we have shown how partial functions can be easily encoded in a CLP language with sets, such as $\{log\}$ [3] (pronounced 'setlog'). Specifically, partial functions can be represented in $\{log\}$ as sets of pairs, where each pair (x, y) is represented as a list of two elements $[x, y]$. Operations on partial functions can be implemented by user-defined predicates in such a way to

enforce the characteristic properties of partial functions over the corresponding set representations. For example, the following predicates implement the domain and range operations:

```
dom({},{}).
dom({[X,Y]/Rel},Dom) :- dom(Rel,D) & Dom = {X/D} & X nin D.
ran({},{}).
ran({[X,Y]/Rel},Ran) :- ran(Rel,R) & Ran = {Y/R} & Y nin R.
```

`dom(F,D)` is true if `D` is the domain of the partial function `F`, whereas `ran(F,R)` is true if `R` is the range of the partial function `F`.[3]

When partial functions are completely specified this approach is satisfactory, at least from an 'operational' point of view. On the other hand, when some elements of a partial function or (part of) the partial function itself are left unspecified—i.e., they are represented by unbound variables—then this approach presents major flaws. For example, the predicate `ran(F,{1})`, where `F` is an unbound variable that represents a partial function, admits infinite distinct solutions `F = {[X1,1]}, F = {[X1,1],[X2,1]}, ...`, that $\{log\}$ computes one after the other though backtracking. If subsequently a failure is detected, such as with the goal `ran(F,{1}) & dom(F,{})`, then the computation loops forever and $\{log\}$ is not able to detect the unsatisfiability.

Making the implementation of predicates over partial functions more sophisticated as shown for instance in [5] may help in solving more efficiently a larger number of cases, but does not provide a completely satisfactory solution in the general case. In fact, there are still cases, such as that considered above, in which there is no simple finite representation of the possibly infinite solutions and this may cause the interpreter to go into infinite computations.

Most of the above mentioned problems could be solved by viewing partial functions as first-class entities of the language and the operations dealing with them as primitive constraints, for which the constraint language provides a suitable solver.

Selecting $\{log\}$ as the host constraint language for this embedding gives one the possibility to exploit its flexible and general management of sets to represent partial functions and to provide many basic set-theoretical operations on partial functions as primitive set constraints for free. Other more specific operations on partial functions can be added to the language as primitive constraints and the solver can be extended accordingly.

The main original results of this work are:

- the identification of a small set of operations on partial functions, to be dealt with as primitive constraints, which are sufficient to represent all other common operations on partial functions as simple conjunctions of these constraints

[3] Notice that in these definitions the `nin` constraints are used to discard those solutions for `Dom` (resp, `Ran`) which contain repeated occurrences of the same element `X`, that might cause the recursive call to `dom` (resp, `ran`) to not terminate.

– the definition of a collection of rewrite rules to simplify conjunctions of primitive constraints
– the definition of a labelling mechanism based on the notion of finite representable domains for partial functions
– the definition of a collection of inference rules to detect possible inconsistencies without the need to perform time-consuming labelling operations.

The rest of this paper is organized as follows. In Section 2, we briefly recall the main features of the language $\{log\}$. The new extended language with partial functions is presented in Section 3, focusing on what is new with respect to $\{log\}$. In Section 4 we describe the constraint rewriting procedures for the new constraints and the global organization of the constraint solver. The labeling mechanism with the introduction of pf-domains is addressed in Section 5. Section 6 introduces a number of inference rules that allow the solver to decide satisfiability of irreducible constraints without having to resort to pf-domains, thus improving its overall efficiency.

2 {log}

$\{log\}$ is a *Constraint Logic Programming (CLP)* language, whose constraint domain is that of *hereditarily finite sets*—i.e., finitely nested sets that are finite at each level of nesting. $\{log\}$ allows sets to be *nested* and *partially specified*—e.g., set elements can contain unbound variables, and it is possible to operate with sets that have been only partially specified. $\{log\}$ provides a collection of primitive constraint predicates, sufficient to represent all the most commonly used set-theoretic operations—e.g., union, intersection, difference.

The $\{log\}$ language was first presented in [2]. A complete constraint solver for the pure CLP fragment included in $\{log\}$—called CLP($\mathcal{SET}$)—is described in [3], while its extension to incorporate intervals and Finite Domain constraints is briefly presented in [4]. Hereafter, with the name CLP($\mathcal{SET}$) we will refer to this last version of our constraint language, while $\{log\}$ will refer to the whole language including CLP($\mathcal{SET}$), along with a number of other syntactic extensions and extra-logical Prolog-like facilities. A working implementation of $\{log\}$ (actually, an interpreter written in Prolog) is available at `http://people.math.unipr.it/gianfranco.rossi/setlog.Home.html`.

Sets are denoted by *set terms*. For example, $\{1, 1, 2\}$, $\{2, 1\}$, and $\{1, 2\}$ are set terms, all denoting the same set of two elements, 1 and 2; $\{X, Y|S\}$ is a set term denoting a partially specified set containing one or two elements, depending on whether X is equal to Y or not, and a, possibly empty, unknown part S.

A *primitive $\mathcal{SET}$-constraint* is defined as any literal based on the set of predicate symbols $\Pi_C = \{=, \mathsf{in}, \mathsf{un}, \mathsf{disj}, \leq, \mathsf{size}, \mathsf{set}, \mathsf{integer}\}$. The predicates $=$ and in represent the equality and the membership relation, respectively. The predicate un represents the union relation: $\mathsf{un}(r, s, t)$ holds if and only if $t = r \cup s$. The predicate disj represents the disjoint relationship between two sets: $\mathsf{disj}(s, t)$ holds if and only if $s \cap t = \emptyset$. The predicate size represents set cardinality: $\mathsf{size}(s, n)$ holds if and only if $n = |s|$. Finally, the predicate $\leq$ represents the

comparison relation "less or equal" over the integer numbers. Predicate symbols neq, nin, nunion, $\mathsf{ninteger}$, ..., are used to denote the negated versions of the corresponding constraint predicates—e.g., s nin t represents the literal $\neg(s$ in $t)$.

Most other useful set-theoretical predicates, e.g., subset (for $\subseteq$) and inters (for $\cap$), can be defined as $\mathcal{SET}$-constraints, using disj and un—e.g., $\mathsf{subset}(u, v) \Leftrightarrow \mathsf{un}(u, v, v)$ [3]. Similarly, other interesting integer predicates (e.g., $<$, $\geq$, and $>$) can be defined as $\mathcal{SET}$-constraints using $\leq$ and $=$.

Example 1. The following formulas are admissible $\mathcal{SET}$-constraints (R, S, T, X, and N are variables):

(i) 1 in $R \wedge 1$ nin $S \wedge \mathsf{inters}(R, S, T) \wedge T = \{X\}$
(ii) $\mathsf{inters}(R, S, T) \wedge \mathsf{size}(T, N) \wedge N =< 2$.

Their informal interpretation is as follows: (i) the set T is the intersection between sets R and S, R must contain 1 and S must not, and T must be a singleton set; (ii) the cardinality of $R \cap S$ must be not greater than 2.

CLP($\mathcal{SET}$) is endowed with a complete *constraint solver*, called $SAT_{\mathcal{SET}}$, for verifying the satisfiability of $\mathcal{SET}$-constraints. Given a constraint C, $SAT_{\mathcal{SET}}(C)$ transforms C either to false (if C is unsatisfiable) or to a finite collection $\{C_1, \ldots, C_k\}$ of constraints in *solved form* [3]. A constraint in solved form is guaranteed to be satisfiable w.r.t. the underlying interpretation structure. Moreover, the disjunction of all the constraints in solved form generated by $SAT_{\mathcal{SET}}(C)$ is equisatisfiable to C in the structure. A detailed description of the constraint solver $SAT_{\mathcal{SET}}$ can be found in [3]. Its implementation is included in the $\{log\}$-interpreter [7].

Example 2. Let C be $\{1, 2 \,|\, X\} = \{1 \,|\, Y\} \wedge 2$ nin X. Then $SAT_{\mathcal{SET}}(C)$ returns, one by one, the following three answers each of which is a constraint in solved form: (i) $Y = \{2 \,|\, X\} \wedge 2$ nin $X \wedge \mathsf{set}(X)$; (ii) $X = \{1 \,|\, N\} \wedge Y = \{2 \,|\, N\} \wedge \mathsf{set}(N) \wedge$ 2 nin N; (iii) $Y = \{1, 2 \,|\, X\} \wedge 2$ nin $X \wedge \mathsf{set}(X)$ (where N is a new variable).

3 The extended language CLP($\mathcal{PF}$)

The constraint domain $\mathcal{SET}$ is extended so as to incorporate partial functions. The new constraint domain and the related language are called $\mathcal{PF}$ and CLP($\mathcal{PF}$), respectively. Since $\mathcal{PF}$ includes $\mathcal{SET}$ as a special case we will simply highlight what is new in $\mathcal{PF}$ with respect to $\mathcal{SET}$.

As concerns syntax, since partial functions can be easily represented as sets, our choice is to not introduce any special symbol to represent them. Partial functions are just a particular kind of sets. Forcing a set to represent a partial function will be obtained at run-time by using suitable constraints on its elements.

Definition 1. *We say that a set term r represents a* partial function *if r has one of the forms:* $\{\}$ *or* $\{[x_1, t_1], [x_2, t_2], \ldots, [x_n, t_n]\}$ *or* $\{[x_1, t_1], [x_2, t_2], \ldots, [x_n, t_n] \mid s\}$*, and x_i, t_i, $i = 1, \ldots, n$, are terms and the constraints $x_i \neq x_j$, $x_i \notin \mathrm{dom}\, s$, hold for all $i, j = 1, \ldots, n$, $i \neq j$.*

A critical issue in the definition of $\mathcal{PF}$ is the choice of which operations over partial functions should be *primitive*—i.e., part of Π_C—and which, on the contrary, should be *programmed* using the language itself. Minimizing the number of predicate symbols in Π_C has the advantage of reducing the number of different kinds of constraints to be dealt with and, hopefully, simplifying the language and its implementation. On the other hand, having to implement such operations on top of the language may lead to efficiency and effectiveness problems, similar to those encountered with the implementation of partial functions using $\{log\}$ discussed in Section 1.

Our choice is to add to the set Π_C of constraint predicate symbols used in $\mathcal{PF}$, a few basic predicate symbols dealing with partial functions which, however, are sufficient to define most of the common operations on partial functions as $(\mathcal{SET}, \mathcal{PF})$-constraints, i.e. as simple conjunctions of primitive $(\mathcal{SET}, \mathcal{PF})$-constraints.

Specifically, we enlarge the set Π_C with the following four predicate symbols:

$$\mathsf{dom}, \mathsf{ran}, \mathsf{comp}, \mathsf{pfun}.$$

The intuitive interpretation of these predicate symbols is: $\mathsf{dom}(r, a)$ (resp. $\mathsf{ran}(r, a)$) holds iff a is the domain (resp., range) of the partial function r; $\mathsf{comp}(r, s, t)$ holds iff the partial function t is the composition of the partial functions r and s, i.e. $t = \{[x, z] : \exists y([x, y] \in r \wedge [y, z] \in t)\}$; $\mathsf{pfun}(r)$ holds iff r is a partial function. Atomic predicates based on these symbols are the only primitive constraints that CLP($\mathcal{PF}$) offers to deal with partial functions (let us simply call these constraints $\mathcal{PF}$-*constraints*). A (general) $(\mathcal{SET}, \mathcal{PF})$-*constraint* is just a conjunction of primitive constraints build using the enlarged Π_C, i.e. $\{=, \mathsf{in}, \mathsf{un}, \mathsf{disj}, \leq, \mathsf{size}, \mathsf{set}, \mathsf{integer}\} \cup \{\mathsf{dom}, \mathsf{ran}, \mathsf{comp}, \mathsf{pfun}\}$.

The following theorem ensures that the primitive $(\mathcal{SET}, \mathcal{PF})$-constraints are sufficient for our purposes. Complete proofs of this and the remaining theorems are available on-line at `http://people.math.unipr.it/gianfranco.rossi/SETLOG/setlogpf_proofs.pdf`). Many of these theorems were proved formally using the Z/EVES proof assistant [8].

Theorem 1. *Literals based on predicate symbols:* dres *(domain restriction),* rres *(range restriction),* ndres *(domain anti-restriction),* nrres *(range anti-restriction),* rimg *(relational image),* oplus *(overriding) and* id *(identity relation) can be replaced by equivalent conjunctions of literals based on* $=$, un, disj, dom, ran *and* comp.

Proof (sketch). The following equivalences hold:

$$\begin{aligned}
\mathsf{ndres}(a, r, s) &\Leftrightarrow \mathsf{dres}(a, r, b) \wedge \mathsf{diff}(r, b, s) \\
\mathsf{nrres}(b, r, s) &\Leftrightarrow \mathsf{rres}(b, r, a) \wedge \mathsf{diff}(r, a, s) \\
\mathsf{dres}(a, r, s) &\Leftrightarrow \mathsf{dom}(r, dr) \wedge \mathsf{dom}(s, ds) \wedge \mathsf{inters}(a, dr, ds) \wedge \mathsf{subset}(s, r) \\
\mathsf{rres}(b, r, s) &\Leftrightarrow \mathsf{un}(s, t, r) \wedge \mathsf{ran}(s, rs) \wedge \mathsf{ran}(r, rr) \\
&\qquad \wedge \mathsf{inters}(b, rr, rs) \wedge \mathsf{ran}(t, rt) \wedge \mathsf{disj}(rs, rt) \\
\mathsf{rimg}(b, r, s) &\Leftrightarrow \mathsf{dres}(b, r, rb) \wedge \mathsf{ran}(rb, s) \\
\mathsf{oplus}(r, s, t) &\Leftrightarrow \mathsf{un}(rs, s, t) \wedge \mathsf{ndres}(ds, r, rs) \wedge \mathsf{dom}(s, ds) \\
\mathsf{id}(a, r) &\Leftrightarrow \mathsf{dom}(r, a) \wedge \mathsf{ran}(r, a) \wedge \mathsf{comp}(r, r, r)
\end{aligned}$$

Other common operations on partial functions can be defined in the same way. For example, the application of a partial function r to an element x can be easily defined in terms of primitive constraints as follows: $\mathsf{apply}(r, x, y)$ is true if and only if $[x, y]$ in r holds.

The ability to express operations on partial functions as $(\mathcal{SET}, \mathcal{PF})$-constraints as stated in Theorem 1 allows us to not consider these operations in the definition of the constraint solver for CLP($\mathcal{PF}$) and to focus our attention only on the four primitive constraints based on pfun, dom, ran and comp.

The selection of these four predicates as primitive constraints is (informally) motivated as follows. Since a function is a tuple of the form (dom, law, ran), then choosing dom and ran seems a rather obvious choice; the *law* can be given as membership predicates (i.e. apply) which already is part of the primitive constraints; pfun is easy to justify since it is necessary to state what sets are partial functions; finally, comp is justified by observing that it is hardly definable in terms of the other primitive constraints. It is worth noting, however, that the proposed subset of primitive predicate symbols is by no way the only possible choice. What we are claiming is that it is a "good" choice, which allows us to define a new solver for partial functions and to master its complexity. Proving that this subset is the minimal one, as well as comparing our choice with other possible choices, in terms of, e.g., expressive power, completeness, effectiveness, and efficiency, is out of the scope of the present work.

4 Constraint Rewriting Procedures

For each primitive constraint symbol $\pi \in \Pi_C$, we develop a *constraint rewriting procedure* specifically devoted to process that type of constraint. Basically, each procedure repeatedly applies to the input constraint C a collection of *rewrite rules* for π until either C becomes false or no rule for π applies to C. At any moment, C represents the *constraint store* managed by the solver.

The rewrite rules have the following general form

$$\frac{\textit{pre-conditions}}{\{C_1, \ldots, C_n\} \rightarrow \{C'_1, \ldots, C'_m\}}$$

where C_i and C'_i are primitive $(\mathcal{SET}, \mathcal{PF})$-constraints and *pre-conditions* are (possibly empty) boolean conditions on the terms occurring in $C_1, \ldots, C_n$. In order to apply the rule, all *pre-conditions* need to be satisfied. $\{C_1, \ldots, C_n\} \rightarrow \{C'_1, \ldots, C'_m\}$ (n, $m \geq 0$) represents the changes in the constraint store caused by the rule application.

Rewrite rules for dom and comp are shown in Figures 1 and 2, respectively. In these figures, $\mathcal{V}$ represents the set of variables, while $\mathsf{empty}(s)$ is an auxiliary predicate which is defined as follows: $s = \emptyset \vee (s = \mathsf{int}(x, y) \wedge x > y)$ (note that, $\neg\mathsf{empty}(s)$ holds also if s is an unbound variable). The whole collection of rewrite rules for dealing with primitive $\mathcal{PF}$-constraints is available on-line at `http://people.math.unipr.it/gianfranco.rossi/SETLOG/setlogpf_rules.pdf`. Rewrite rules for all other primitive constraints can be found in [3] and [4].

$$\frac{r \in \mathcal{V}}{\{\mathsf{dom}(r, r)\} \rightarrow \{r = \emptyset\}} \tag{1}$$

$$\frac{\mathsf{empty}(a)}{\{\mathsf{dom}(r, a)\} \rightarrow \{r = \emptyset\}} \tag{2}$$

$$\frac{\mathsf{empty}(r)}{\{\mathsf{dom}(r, a)\} \rightarrow \{a = \emptyset\}} \tag{3}$$

$$\frac{r = \{[x, y]|rr\} \quad \neg\mathsf{empty}(a)}{\{\mathsf{dom}(r, a)\} \rightarrow \{a = \{x|rs\}, [x, y] \ \mathsf{nin} \ rr, \mathsf{dom}(rr, rs)\}} \tag{4}$$

$$\frac{r \in \mathcal{V} \quad a = \{x|rs\}}{\{\mathsf{dom}(r, a)\} \rightarrow \{r = \{[x, y]|rr\}, x \ \mathsf{nin} \ rs, \mathsf{dom}(rr, rs)\}} \tag{5}$$

Fig. 1. Rewrite rules for dom constraints

$$\frac{\mathsf{empty}(r)}{\{\mathsf{comp}(r, s, q)\} \rightarrow \{q = \emptyset\}} \tag{10}$$

$$\frac{\mathsf{empty}(s) \quad \neg\mathsf{empty}(r)}{\{\mathsf{comp}(r, s, q)\} \rightarrow \{q = \emptyset\}} \tag{11}$$

$$\frac{\mathsf{empty}(q) \quad \neg\mathsf{empty}(r) \quad \neg\mathsf{empty}(s)}{\{\mathsf{comp}(r, s, q)\} \rightarrow \{\mathsf{ran}(r, rr), \mathsf{dom}(s, ds), \mathsf{disj}(rr, ds)\}} \tag{12}$$

$$\frac{q = \{[x, z]|rq\} \quad \neg\mathsf{empty}(r) \quad \neg\mathsf{empty}(s)}{\begin{array}{c}\{\mathsf{comp}(r, s, q)\} \rightarrow \{r = \{[x, y]|rr\}, \\ s = \{[y, z]|rs\}, [x, z] \ \mathsf{nin} \ rq, [y, z] \ \mathsf{nin} \ rs, \mathsf{comp}(rr, s, rq)\}\end{array}} \tag{13}$$

$$\frac{q \in \mathcal{V} \quad r = \{[x, y]|rr\} \quad \neg\mathsf{empty}(s) \quad s \notin \mathcal{V}}{\begin{array}{c}\{\mathsf{comp}(r, s, q)\} \rightarrow \{s = \{[y, z]|rs\}, \\ q = \{[x, z]|rq\}, [x, y] \ \mathsf{nin} \ rr, [y, z] \ \mathsf{nin} \ rs, \mathsf{comp}(rr, s, rq)\} \\ \mathsf{or} \\ \{\mathsf{comp}(r, s, q)\} \rightarrow \{\mathsf{dom}(s, ds), y \ \mathsf{nin} \ ds, [x, y] \ \mathsf{nin} \ rr, \\ \mathsf{comp}(rr, s, q)\}\end{array}} \tag{14}$$

Fig. 2. Rewrite rules for comp constraints

Rewrite rule (14) is a nondeterministic rule: if the preconditions are met, the rule nondeterministically performs one of the two rewritings in its lower part. Specifically, rule (14) deals with the case in which both r and s in $\mathsf{comp}(r, s, q)$ are not variables nor empty partial functions. The nondeterministic choice takes care of the fact that, for each pair $[x, y]$ in r, there may exist a z such that $[y, z] \in s$ or there may not exist any z such that $[y, z] \in s$. This last condition is expressed by stating that $\mathsf{dom}(s, ds) \wedge y \ \mathsf{nin} \ ds$.

The global organization of the solver for the new language—called $SAT_{\mathcal{PF}}$—is shown in Algorithm 1. It makes use of two procedures: infer and STEP. infer is used to automatically add the constraints set, integer, and pfun to the constraint C in order to force arguments of primitive constraints to be of the proper type. For example, if C contains the constraint $\mathsf{dom}(r, a)$ then $\mathsf{infer}(C)$ will add to C the constraint $\mathsf{pfun}(r) \wedge \mathsf{set}(a)$. The procedure STEP is the core part of $SAT_{\mathcal{PF}}$: it applies specialized constraint rewriting procedures to the current constraint C and returns the modified constraint. The execution of STEP is iterated until a fixpoint is reached—i.e., the constraint cannot be simplified any further.

Algorithm 1 The CLP($\mathcal{PF}$) Constraint Solver

```
procedure SAT_PF(C)
    C ← infer(C)
    repeat
        C' ← C;
        C ← STEP(C);
    until C = C';
    return C
end procedure
```

When no rewrite rule applies to the considered $\mathcal{PF}$-constraint then the corresponding rewriting procedure terminates immediately and the constraint store remains unchanged. Since no other rewriting procedure deals with the same kind of constraints, the irreducible constraints will be returned as part of the constraint computed by $SAT_{\mathcal{PF}}$. Precisely, if X and X_i are variables and t is a term (either variable or not), the following $\mathcal{PF}$-constraints are dealt with as irreducible:

1. $\mathsf{dom}(X_1, X_2)$, where X_1 and X_2 are distinct variables;
2. $\mathsf{ran}(X, t)$, where t is distinct from X and t is not the empty set;
3. $\mathsf{comp}(X_1, t, X_3)$ or $\mathsf{comp}(t, X_2, X_3)$, where t is not the empty set;
4. $\mathsf{pfun}(X)$ and there are no constraints of the form $\mathsf{integer}(X)$ in C.

For all other primitive $(\mathcal{SET}, \mathcal{PF})$-constraints, $SAT_{\mathcal{PF}}$ uses the rewriting rules of CLP($\mathcal{SET}$) and the irreducible form constraints it returns are all $\mathcal{SET}$-constraints in solved form (cf. Sect. 2 and [3]). Observe that a constraint composed of only solved form literals is proved to be always satisfiable.

Example 3. Constraint rewriting.

- $\mathsf{dom}(\{[a,1],[b,2],[c,1]\},D)$ is rewritten to $D=\{a,b,c\}$
- $\mathsf{ran}(\{[a,1],[b,2],[c,1]\},D)$ is rewritten to $R=\{1,2\}$
- $\mathsf{dom}(\{[a,1]\},\{b\})$ is rewritten to false (actually, it is first rewritten to a constraint containing $\{b\}=\{a|R\}$ which in turn is rewritten to false by the equality rewriting procedure of $SAT_{\mathcal{SET}}$)
- $\mathsf{dom}(\{[1,a]|S\},D)$ is rewritten to either $D=\{1|DR\}\wedge\mathsf{pfun}(S)\wedge\mathsf{dom}(S,DR)\wedge 1\ \mathsf{nin}\ DR\wedge\mathsf{set}(DR)$ or $S=\{[1,a]|SR\}\wedge D=\{1|DR\}\wedge\mathsf{pfun}(SR)\wedge\mathsf{dom}(SR,DR)\wedge 1\ \mathsf{nin}\ DR\wedge\mathsf{set}(DR)$
- $\mathsf{comp}(\{[1,b]\},B,\{[1,a]\})$ is rewritten to $B=\{[b,a]|BR\}\wedge[b,a]\ \mathsf{nin}\ BR\wedge\mathsf{pfun}(BR)\wedge\mathsf{dom}(BR,D)\wedge b\ \mathsf{nin}\ D\wedge\mathsf{set}(D)$
- $\mathsf{inters}(\{X\},\{1\},D)\wedge\mathsf{dom}(R,D)\wedge\mathsf{ran}(R,\emptyset)$ is rewritten to $D=\emptyset\wedge R=\emptyset\wedge X\ \mathsf{neq}\ 1$
- $\mathsf{apply}(F,X,Y)\wedge\mathsf{dom}(F,D)\wedge X\ \mathsf{nin}\ D$ is rewritten to false.

Note that with the implementation of dom and ran as user-defined $\{log\}$ predicates (see [5]) the last two goals would loop forever.

The $SAT_{\mathcal{PF}}$ procedure is proved to be always terminating.

Theorem 2 (Termination). *The $SAT_{\mathcal{PF}}$ procedure terminates for every input constraint C.*

The termination of $SAT_{\mathcal{PF}}$ and the finiteness of the number of non-deterministic choices generated during its computation, guarantee the finiteness of the number of constraints non-deterministically returned by $SAT_{\mathcal{PF}}$. Therefore, $SAT_{\mathcal{PF}}$ applied to a constraint C always terminates, rewriting C to either false or to a (finite) disjunction of $(\mathcal{SET},\mathcal{PF})$-constraints in a simplified form. The following theorem proves that the collection of constraints in irreducible form generated by $SAT_{\mathcal{PF}}$ preserves the set of solutions of the input constraint.

Theorem 3 (Soundness and Completeness). *Let C be a constraint, C_1, ..., C_n be the constraints obtained from $SAT_{\mathcal{PF}}(C)$, σ be a valuation of C and $C_1\vee\ldots\vee C_n$, expanded to the new variables possibly introduced into $C_1,\ldots,C_n$ by the rewrite procedures, and $\mathcal{A}_{\mathcal{PF}}$ be the interpretation structure associated with the constraint domain $\mathcal{PF}$. Then, $\mathcal{A}_{\mathcal{PF}}\models\sigma(C)$ if and only if $\mathcal{A}_{\mathcal{PF}}\models\sigma(C_1\vee\ldots\vee C_n)$.*

If at least one of the constraint C_i returned by $SAT_{\mathcal{PF}}(C)$ contains *only* primitive $\mathcal{SET}$-constraints then, according to [3], C_i is in solved form and it is surely satisfiable. Therefore, in this case, thanks to Theorems 2 and 3, we can conclude that the original constraint C is surely satisfiable.

Unfortunately, this is not always the case, as discussed in the next section.

5 pf-domains

Differently from CLP($\mathcal{SET}$), the simplified constraint returned by $SAT_{\mathcal{PF}}$ is not guaranteed to be satisfiable.

Example 4. The following $(\mathcal{SET}, \mathcal{PF})$-constraint

$$\mathsf{dom}(R, D) \wedge R \ \mathsf{neq}\ \emptyset \wedge \mathsf{un}(D, Y, Z) \wedge \mathsf{disj}(D, Z)$$

is an irreducible constraint but it is clearly unsatisfiable (the only possible solution for $\mathsf{un}(D, Y, Z) \wedge \mathsf{disj}(D, Z)$ is $D = \emptyset$, and $D = \emptyset$ if and only if $R = \emptyset$).

Thus, differently from CLP($\mathcal{SET}$), the ability to produce a collection of constraints in an irreducible form from the input constraint C cannot be used to decide the satisfiability of C. As many concrete solvers, e.g. the CLP($\mathcal{FD}$) solvers, $SAT_{\mathcal{PF}}$ is an *incomplete* solver. Thus, if it returns false the input constraint is surely unsatisfiable, whereas if it returns a constraint in irreducible form then we cannot conclude that the input constraint is surely satisfiable.

In order to obtain a complete solver, we provide a way to associate a *finitely representable domain* to each partial function variable and to force these variables to get values from their associated domains, i.e. to perform *labeling* on them. This is obtained by defining a new constraint pfun, of arity 2, with the following interpretation:

$$\mathsf{pfun}^{\mathcal{S}}(r, n) \text{ if and only if } r \in X \nrightarrow Y \wedge n \in \mathbb{N} \wedge |r| \leq n$$

The solutions of $\mathsf{pfun}(r, n)$ are all the partial functions r with cardinality less or equal to n. The ability to represent domains and ranges of partial functions as partially specified sets, i.e. sets containing unbound variables as their elements, allows us to provide a *finite* representation for the (possibly infinite) set of all solutions of $\mathsf{pfun}(r, n)$. For example, the set of solutions for $\mathsf{pfun}(r, 2)$, where r is a variable, can be represented by the following equisatisfiable disjunction of three primitive constraints: $r = \emptyset \vee r = \{[X, Y]\} \vee r = \{[X_1, Y_1], [X_2, Y_2]\} \wedge X_1 \ \mathsf{neq}\ X_2$.

We will call the set of partial functions represented by these constraints the *pf-domain* of the pf-variable r. pf-domains represent in general infinite sets but they are finitely representable in our language. Notice that the $\mathsf{pfun}/2$ constraints require to specify an *upper bound* for the cardinality of the involved partial functions, not to fix its exact value. In this sense, it is not a real restriction to the expressive power of the language.

From an operational point of view, solving $\mathsf{pfun}(r, n)$, with n a constant natural number, non-deterministically computes, one after the other, all the $n+1$ possible assignments for r. Therefore, solving $\mathsf{pfun}(r, n)$ allows us to perform a sort of *labeling* for the pf-variable r. Notice that, differently from $\mathsf{pfun}(r)$, $\mathsf{pfun}(r, n)$ has no irreducible form. If r is an unbound variable (n is required to be a constant number), then solving $\mathsf{pfun}(r, n)$ always generates an equality for r, along with possible inequality constraints over the elements in the domain of r.

The labeling process involved in $\mathsf{pfun}/2$ constraints do not compromise termination of the procedure $SAT_{\mathcal{PF}}$ since the set of possible values to be assigned to partial function variables through labeling is anyway finite. Moreover, assuming our domain of discourse is limited to finite partial functions only, it is straightforward to see that the rewriting rules for $\mathsf{pfun}/2$ preserve the set of solutions of

the input constraint. Thus we can immediately extend to $\mathsf{pfun}/2$ constraints the results of Theorems 2 and 3.

Solving $\mathsf{pfun}/2$ constraints allows pf-variables to always get a value, although it can be a non-ground value. This is enough, however, to guarantee that all $\mathcal{PF}$-constraints are completely eliminated at the end of the computation.

Lemma 1. *Let C be the input constraint and $V_1, \ldots, V_n$ all the pf-variables occurring in C. If C contains $\mathsf{pfun}(V_1, k_1) \wedge \cdots \wedge \mathsf{pfun}(V_n, k_n)$, $k_1, \ldots, k_n \in \mathbb{N}$, then $SAT_{\mathcal{PF}}(C)$ returns either* false *or a disjunction of $(\mathcal{SET})$-constraints in* solved form*.*

Remembering that $\mathcal{SET}$-constraints in solved form are always satisfiable, Lemma 1 guarantees that, if the input constraint C contains $\mathsf{pfun}/2$ constraints for all the pf-variables occurring in it and $SAT_{\mathcal{PF}}(C)$ does not terminate with false, then the disjunction of constraints returned by $SAT_{\mathcal{PF}}(C)$ is surely satisfiable. Thanks to soundness and completeness of $SAT_{\mathcal{PF}}$ extended to $\mathsf{pfun}/2$ constraints, we can conclude that in this case C is satisfiable.

Hence, by properly exploiting $\mathsf{pfun}/2$ constraints, we get a complete solver. This means that our solver can detect all cases in which the input constraint is unsatisfiable, as well as all cases in which the input constraint is satisfiable and, in these cases, it can generate all viable solutions.

Example 5. The following constraints are all rewritten to either false or to a constraint in solved form, whereas they are simply left unchanged if no pf-domain is specified.

- $\mathsf{ran}(X, \{1\}) \wedge \mathsf{un}(X, Y, Z) \wedge \mathsf{pfun}(X, 100)$ is rewritten to the solved form constraints (first two solutions): $X = \{[A, 1]\} \wedge Z = \{[A, 1]|Y\} \wedge \mathsf{set}(Y)$; $X = \{[A, 1], [B, 1]\} \wedge Z = \{[A, 1], [B, 1]|Y\} \wedge \mathsf{set}(Y) \wedge A \ \mathsf{neq}\ B$
- $\mathsf{ran}(Z1, R) \wedge \mathsf{ran}(Z2, R) \wedge \mathsf{dom}(Z1, S) \wedge \mathsf{dom}(Z2, S) \wedge Z1 \ \mathsf{neq}\ Z2 \wedge \mathsf{pfun}(Z1, 100)$ is rewritten to the solved form constraint (first solution): $Z1 = \{[A, B], [C, D]\} \wedge Z2 = \{[A, D], [C, B]\} \wedge R = \{B, D\} \wedge S = \{A, C\} \wedge B \ \mathsf{neq}\ D \wedge A \ \mathsf{neq}\ C$
- $\mathsf{dom}(R, D) \wedge D \ \mathsf{neq}\ \emptyset \wedge \mathsf{un}(D, Y, Z) \wedge \mathsf{disj}(D, Z) \wedge \mathsf{pfun}(R, 5)$ is rewritten to false
- $\mathsf{comp}(\{[1, a]\}, Y, Z) \wedge \mathsf{dom}(Y, S) \wedge a \ \mathsf{nin}\ S \wedge Z \ \mathsf{neq}\ \emptyset \wedge \mathsf{pfun}(Y, 5)$ is rewritten to false.

Finally, it is worth noting that, while having to specify domain information through the $\mathsf{pfun}/2$ constraints may be cumbersome in some cases, it can be of great importance in other cases. For example, in the application described in [5] where $\{log\}$ is used as a test case generator it may be important to be able to generate possible values (actually, just one solution is enough in this case) for the variables occurring in goals that are proved to be satisfiable.

6 Improving constraint solving

From a more practical point of view, having to perform labeling for pf-variables, may cause unacceptable execution time in some cases. For example, the con-

straint

$$\mathsf{dom}(R, D1) \wedge \mathsf{dom}(R, D2) \wedge D1 \ \mathsf{neq} \ D2 \wedge \mathsf{pfun}(R, k)$$

is proved to be unsatisfiable, but only for relatively small values of k.

To alleviate this problem, we introduce a number of new rewrite rules—hereafter simply called *inference rules*—that allow new constraints to be inferred from the irreducible constraints. The presence of these additional constraints allows the solver to deduce possible unsatisfiability of the given constraint without having to resort to any labeling process, thus improving the overall efficiency of constraint solving in many cases.

The inference rules are applied by calling function infer_rules just after the iteration of STEP ends finding a fixpoint (see Algorithm 1). infer_rules(C) applies all possible inference rules to all possible primitive constraints in C. After the rules have been applied, possibly modifying C, the STEP loop is repeated from the beginning. Only when both STEP and infer_rules do not modify C, then the new global constraint solving procedure—called $SAT'_{\mathcal{PF}}$—ends.

Each inference rule captures some property of the primitive operators for partial functions, possibly relating these operators with other general operators, such as inequality (constraint neq) and set cardinality (constraint size). All rules take into account one or two primitive constraints at a time and add new primitive constraints to the constraint store. Figure 3 shows the inference rules which make sure that the domain and range of a partial function are unique, while Figure 4 shows the rules expressing the not emptiness relation among partial functions and their domains and ranges. The whole collection of inference rules used by $SAT'_{\mathcal{PF}}$ is available on-line at `http://people.math.unipr.it/gianfranco.rossi/SETLOG/setlogpf_rules.pdf`).

$$\frac{}{\{\mathsf{dom}(r, a), \mathsf{dom}(r, b)\} \rightarrow \{\mathsf{dom}(r, a), a = b\}} \quad (21)$$

$$\frac{}{\{\mathsf{ran}(r, a), \mathsf{ran}(r, b)\} \rightarrow \{\mathsf{ran}(r, a), a = b\}} \quad (22)$$

Fig. 3. Inference rules for domain and range uniqueness

$$\frac{}{\{\mathsf{dom}(r,a), a \ \mathsf{neq}\ \emptyset\} \rightarrow \{\mathsf{dom}(r,a), a \ \mathsf{neq}\ \emptyset, r \ \mathsf{neq}\ \emptyset\}} \quad (23)$$

$$\frac{}{\{\mathsf{dom}(r,a), r \ \mathsf{neq}\ \emptyset\} \rightarrow \{\mathsf{dom}(r,a), r \ \mathsf{neq}\ \emptyset, a \ \mathsf{neq}\ \emptyset\}} \quad (24)$$

$$\frac{}{\{\mathsf{ran}(r,a), a \ \mathsf{neq}\ \emptyset\} \rightarrow \{\mathsf{ran}(r,a), a \ \mathsf{neq}\ \emptyset, r \ \mathsf{neq}\ \emptyset\}} \quad (25)$$

$$\frac{a \in \mathcal{V}}{\{\mathsf{ran}(r,a), r \ \mathsf{neq}\ \emptyset\} \rightarrow \{\mathsf{ran}(r,a), r \ \mathsf{neq}\ \emptyset, a \ \mathsf{neq}\ \emptyset\}} \quad (26)$$

$$\frac{a \notin \mathcal{V}}{\{\mathsf{ran}(r,a)\} \rightarrow \{\mathsf{ran}(r,a), r \ \mathsf{neq}\ \emptyset\}} \quad (27)$$

Fig. 4. Inference rules for domain and range not emptiness

Example 6. The following constraints are all proved to be unsatisfiable using $SAT'_{\mathcal{PF}}$ (rule numbers refer to the on-line document where all rules are listed):

$\mathsf{dom}(X, D1) \wedge \mathsf{dom}(X, D2) \wedge D1 \ \mathsf{neq}\ D2$ (using rule (21))
$\mathsf{ran}(X, \{1\}) \wedge \mathsf{ran}(X, \{A\}) \wedge A \ \mathsf{neq}\ 1$ (using rule (22))
$\mathsf{dom}(X, DX) \wedge X \ \mathsf{neq}\ \emptyset \wedge \mathsf{disj}(DX, Z) \wedge \mathsf{un}(DX, Y, Z)$ (using rule (24))
$\mathsf{ran}(X, RX) \wedge RX \ \mathsf{neq}\ \emptyset \wedge \mathsf{disj}(X, Z) \wedge \mathsf{un}(X, Y, Z)$ (using rule (26))
$\mathsf{dom}(X, DX) \wedge \mathsf{size}(X, N) \wedge \mathsf{size}(DX, M) \wedge N \ \mathsf{neq}\ M$ (using rule (28))
$\mathsf{comp}(\{[a, 1]\}, Y, Z) \wedge \mathsf{dom}(Z, DZ) \wedge a \ \mathsf{nin}\ DZ$
$\wedge Z \ \mathsf{neq}\ \emptyset$ (using rule (30))
$\mathsf{un}(X, Y, Z) \wedge \mathsf{dom}(X, D) \wedge \mathsf{dom}(Y, D)$
$\wedge \mathsf{dom}(Z, DZ) \wedge D \ \mathsf{neq}\ DZ$ (using rule (33))

The same constraints of Example 6 but using $SAT_{\mathcal{PF}}$, that is without applying any inference rule, are simply treated as irreducible. On the other hand, adding constraints $\mathsf{pfun}/2$ to perform labeling on pf-variables would allow $SAT_{\mathcal{PF}}$ to detect the unsatisfiability for all these constraints, but only when the specified partial function cardinalities are relatively small the response times would be practically acceptable.

Termination of the improved constraint solver is stated by the following theorem.

Theorem 4 (Termination of $SAT'_{\mathcal{PF}}$). *The $SAT'_{\mathcal{PF}}$ procedure can be implemented in such a way that it terminates for every input constraint C.*

Soundness and completeness of the extended solver $SAT'_{\mathcal{PF}}$ come from soundness and completeness of $SAT_{\mathcal{PF}}$ and from the following theorem, which ensures that the added constraints do not modify the set of solutions of the original constraint.

Theorem 5 (Equisatisfiability of inference rules). *Let S be a constraint and S' be the constraint obtained from the inference rules* (21)–(34)*. Then S' is equisatisfiable to S with respect to the interpretation structure $\mathcal{A}_{\mathcal{PF}}$.*

$SAT'_{\mathcal{PF}}$ is still not a complete solver. As a counterexample, consider the following constraint

$$\mathsf{ran}(X, \{1\}) \wedge \mathsf{ran}(Y, \{1, 2\}) \wedge \mathsf{dom}(X, D) \wedge \mathsf{dom}(Y, D) \wedge \mathsf{disj}(X, Y).$$

This constraint is unsatisfiable with respect to $\mathcal{A}_{\mathcal{PF}}$, but $SAT'_{\mathcal{PF}}$ is not able to prove this fact (it simply leaves the constraint unchanged).

New inference rules could be defined and added to the solver to detect further properties of the partial function domain, thus avoiding as much as possible the need for pfun/2 constraints. For example, the following inference rule relates comp, size and ran:

$$\frac{}{\{\mathsf{comp}(r, s, q)\} \rightarrow \{\mathsf{comp}(r, s, q), \mathsf{ran}(q, a), \mathsf{size}(a, n), \mathsf{size}(s, m), n \leq m\}}$$

However, finding a collection of inference rules that guarantees us to obtain a complete solver, regardless of the presence of pfun/2 constraints, seems to be a difficult task. Moreover, checking the constraint store to detect applicable inference rules may be quite costly in general. Thus, the solution we adopted is based on finding a tradeoff between efficiency and completeness, as usual in many concrete constraint solvers. Only those properties that require relatively small effort to be checked are taken into account by the solver. For all cases not covered by the inference rules, however, solver's completeness is obtained by exploiting pf-domains and pfun/2 constraints. Further empirical assessment of the solver may lead to review the current choices and provide additional inference rules in future releases.

7 Conclusions

In this paper we have shown how to integrate partial functions as first-class citizens into the constraint logic programming language with sets $\{log\}$. Since partial functions can be viewed as sets, they are embedded quite smoothly into $\{log\}$, and all facilities for set manipulation offered by $\{log\}$ are immediately available to manipulate partial functions as well. We have added to the language a very limited number of new primitive constraints specifically devoted to deal with partial functions and we have provided sound, complete and terminating rewriting procedures for them. The resulting constraint solver either terminates with false or with a disjunction of simplified constraints which the solver cannot further simplify (i.e., irreducible constraints). We have identified conditions under which the ability to generate such a disjunction guarantees the satisfiability of the input constraint. Moreover, we have defined a number of inference rules that allow the solver to detect, in many cases, unsatisfiability even in the more general situations (e.g. without requiring to specify an upper bound for the cardinality of partial functions).

For the future, there are two main correlated lines of work:

– identifying more precisely the class of irreducible constraints which are guaranteed to be satisfiable; so far this class is restricted to irreducible constraints not containing pf-constraints, but it is likely to be enlarged to include pf-constraints as well, at least of some specific form (e.g., those which contain only unbound variables, thus excluding for instance the irreducible constraints of the form $\mathsf{ran}(X, \{\dots\})$)
– defining new inference rules that allow further "hidden" properties of irreducible constraints to be made explicit, in order to make constraint solving more and more "precise"; that is, on the one hand, to allow the solver to detect more and more unsatisfiable constraints and, on the other hand, to allow the class of irreducible constraints whose satisfiability can be decided without the need to perform any labeling operation to be enlarged as much as possible.

References

1. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA (1996).
2. Dovier, A., Omodeo, E., Pontelli, E., Rossi, G.: A Language for Programming in Logic with Finite Sets. *J. Log. Program.* 28, 1 (1996), 1–44.
3. Dovier, A., Piazza, C., Pontelli, E., Rossi, G.: Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst.* 22, 5 (2000), 861–931.
4. Dal Palù, A., Dovier, A., Pontelli, E., Rossi, G.: Integrating finite domain constraints and CLP with sets. In *PPDP*, ACM (2003), 219–229.
5. Cristiá, M., Rossi, G., Frydman, C. S.: {log} as a Test Case Generator for the Test Template Framework. In *SEFM*, Hierons, R. M., Merayo, M. G., Bravetti M. (Eds.), LNCS Vol. 8137, Springer (2013), 229–243.
6. Jackson, D.: Alloy: A logical modelling language. In: *ZB 2003: Formal Specification and Development in Z and B*, Bert, D., Bowen, J.P., King, S., Waldén, M.A. (eds.), LNCS Vol. 2651, Springer (2003).
7. Rossi, G.: {*log*}. (2008). `http://www.math.unipr.it/~gianfr/setlog.Home.html` last access: December 2013.
8. Saaltink, M.: The Z/EVES mathematical toolkit version 2.2 for Z/EVES version 1.5. Technical report, ORA Canada (1997)
9. Spivey, J.M.: The Z notation: a reference manual. Prentice Hall International (UK) Ltd., Hertfordshire, UK (1992).