Adding Partial Functions to Constraint Logic Programming with Sets

MAXIMILIANO CRISTIÁ

CIFASIS and UNR, Rosario, Argentina
(*e-mail:* `cristia@cifasis-conicet.gov.ar`)

GIANFRANCO ROSSI

Università degli Studi di Parma, Parma, Italy
(*e-mail:* `gianfranco.rossi@unipr.it`)

CLAUDIA FRYDMAN

Aix Marseille Univ., CNRS, ENSAM, Univ. de Toulon, LSIS UMR 7296, France
(*e-mail:* `claudia.frydman@lsis.org`)

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. Efficiency of the new version is empirically assessed by running a number of non-trivial set-theoretical goals involving partial functions, obtained from specifications written in Z.

KEYWORDS: CLP, {*log*}, set theory, 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 they may be undefined for some elements in the domain—i.e. they are partial. Binary relations are in turn just sets of ordered pairs. Then, all relational operators (such as dom, ran, $\mathbin{\mathring{,}}$, 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 a partial function f then $f(x)$ is defined as the element, y, in the range of f such that $(x, y) \in f$.

The motivation for adding partial functions to specification/programming languages is primarily to enhance the language's expressive power. In fact, partial functions constitute a powerful and convenient data abstraction. As an example, the relation between the key of a table and the rest of its columns is naturally modeled as a partial function. Partial functions are common in formal specification notations, such as Z (Spivey 1992), B (Abrial 1996) and Alloy (Jackson 2003), which are mainly used to specify state-based systems (notice that, many concepts or features of these systems are best represented as partial functions, not as total functions). Usefulness of partial functions in executable programming languages is attested by the common presence of library facilities, e.g. the `map` class of Java and C++, that support at some extent the partial function abstraction. Availability of maps, dictionaries or similar associative data structures as *primitive* components of some programming languages, such as SETL (Schwartz et al. 1986) or Phyton, also attests usefulness of the partial function abstraction.

Partial functions (or maps or, more generally, binary relations) can be added naturally also to CLP languages with sets, as observed for instance in (Gervet 2006). In particular, in (Cristiá et al. 2013) we have shown how partial functions can be encoded in the CLP language with sets $\{log\}$ (pronounced 'setlog') (Dovier et al. 2000). 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.

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})`, which holds if `{1}` is the range of the partial function `F`, admits infinite distinct solutions `F = {[X1,1]}`, `F = {[X1,1],[X2,1]}`, ..., whenever `F` is unbound. 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 Cristiá et al. (2013) 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. Hence, the motivation for managing partial functions through constraint solving is primarily to enhance the language effectiveness, that is the ability to compute the

satisfiability/unsatisfiability of as many as possible (complex) set-based formulas involving partial functions. 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: (*i*) 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; (*ii*) the definition of a collection of rewrite rules to simplify conjunctions of primitive constraints; (*iii*) the definition of a labeling mechanism based on the notion of finite representable domains for partial functions; (*iv*) the definition of a collection of inference rules to detect possible inconsistencies without the need to perform time-consuming labeling operations.

At our knowledge, only very few works have addressed the problem of adding partial functions as primitive entities in a C(L)P setting. For instance, the Conjunto language (Gervet 1997) provides relation variables at the language level. However, the domain and the range of the relations are limited to ground finite sets. Map variables where the domain and range of the mapping can be also finite set variables are introduced in CP(Map) (Deville et al. 2005). All these proposals, however, do not consider the more general case of partially specified partial functions—where some elements of the domain or the range can be left unknown—which on the contrary are essential in our proposal. Moreover, the collection of primitive constraints on map variables they provide is usually restricted to very few constraints, in particular to model the function application operation.

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. A practical assessment of the performance of the new solver is provided in Section 7.

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 con-

straint predicates, sufficient to represent all the most commonly used set-theoretic operations—e.g., union, intersection, difference.

The $\{log\}$ language was first presented by Dovier et al. (1996). A complete constraint solver for the pure CLP fragment included in $\{log\}$—called CLP($\mathcal{SET}$)—is described by Dovier et al. (2000), while its extension to incorporate intervals and Finite Domain constraints is briefly presented by Dal Palù et al. (2003). 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 on the web (Rossi 2008).

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 $\Pi_C = \{=, \text{in}, \text{un}, \text{disj}, \leq, \text{size}, \text{set}, \text{integer}\}$. Symbols in Π_C have their natural set-theoretic interpretation. In particular, the predicate un represents the union relation ($\text{un}(r, s, t)$ holds if and only if $t = r \cup s$), while the predicate disj represents the disjoint relation between two sets ($\text{disj}(s, t)$ holds if and only if $s \cap t = \emptyset$). Most other useful set-theoretical predicates, e.g., subset and inters, can be defined as $\mathcal{SET}$-constraints, using disj and un—e.g., $\text{subset}(u, v) \Leftrightarrow \text{un}(u, v, v)$ (Dovier et al. 2000). As an example, the following formula, $\text{inters}(R, S, T) \wedge \text{size}(T, N) \wedge N =< 2$, is an admissible $\mathcal{SET}$-constraint whose (informal) interpretation is: 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*. 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 Dovier et al. (2000).

Example 1
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: $Y = \{2|X\} \wedge 2$ nin $X \wedge \text{set}(X)$; $X = \{1|N\} \wedge Y = \{2|N\} \wedge \text{set}(N) \wedge 2$ nin N; and $Y = \{1, 2|X\} \wedge 2$ nin $X \wedge \text{set}(X)$ (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, our choice is to not introduce any special symbol to represent partial functions, since they can be easily represented as sets. 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, s is a set term representing a partial function, and the constraints $x_i \neq x_j$, $x_i \notin \operatorname{dom} s$, 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 extend the set Π_C of constraint predicate symbols 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. $t = \{[x,z] : \exists y([x,y] \in r \wedge [y,z] \in s)\}$; $\mathsf{pfun}(r)$ holds iff r is a partial function.

Atomic predicates based on these symbols are the only primitive constraints that $\mathrm{CLP}(\mathcal{PF})$ 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 built 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 constraints are sufficient to define most of the common operations on partial functions as $(\mathcal{SET}, \mathcal{PF})$-constraints. 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 formally proved using the Z/EVES proof assistant (Saaltink 1997).

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) can be replaced by equivalent conjunctions of literals based on $=$, un, disj, dom, ran and comp.

Proof (sketch)
The following equivalences hold:

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

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 $\mathrm{CLP}(\mathcal{PF})$ and to focus our attention only on the four primitive constraints based on pfun, dom, ran and comp.

It is worth noting that the proposed subset of primitive predicate symbols is not the only possible choice. Roughly speaking, it is motivated by observing that: since a function is a tuple of the form (dom, law, ran), then choosing dom and ran seems a rather natural choice; the *law* can be given as membership predicates (i.e. apply) which is already part of the primitive constraints; pfun is easy to justify since it is necessary to state which sets are partial functions; finally, comp is justified by observing that it is hardly definable in terms of the other primitive constraints. However, proving that this subset of primitive constraints 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

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

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. $\{C_1, \ldots, C_n\} \to \{C'_1, \ldots, C'_m\}$
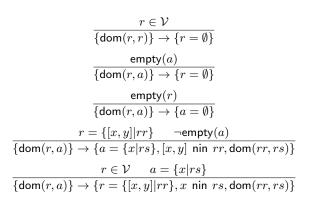
$$\frac{r \in \mathcal{V}}{\{\mathsf{dom}(r,r)\} \to \{r = \emptyset\}}$$

$$\frac{\mathsf{empty}(a)}{\{\mathsf{dom}(r,a)\} \to \{r = \emptyset\}}$$

$$\frac{\mathsf{empty}(r)}{\{\mathsf{dom}(r,a)\} \to \{a = \emptyset\}}$$

$$\frac{r = \{[x,y]|rr\} \qquad \neg\mathsf{empty}(a)}{\{\mathsf{dom}(r,a)\} \to \{a = \{x|rs\}, [x,y] \; \mathsf{nin} \; rr, \mathsf{dom}(rr,rs)\}}$$

$$\frac{r \in \mathcal{V} \qquad a = \{x|rs\}}{\{\mathsf{dom}(r,a)\} \to \{r = \{[x,y]|rr\}, x \; \mathsf{nin} \; rs, \mathsf{dom}(rr,rs)\}}$$

Fig. 1. Rewrite rules for dom.

$$\frac{\mathsf{empty}(q) \qquad \neg\mathsf{empty}(r) \qquad \neg\mathsf{empty}(s)}{\{\mathsf{comp}(r,s,q)\} \to \{\mathsf{ran}(r,rr), \mathsf{dom}(s,ds), \mathsf{disj}(rr,ds)\}}$$

$$\frac{q = \{[x,z]|rq\} \qquad \neg\mathsf{empty}(r) \qquad \neg\mathsf{empty}(s)}{\{\mathsf{comp}(r,s,q)\} \to \{r = \{[x,y]|rr\},}$$
$$s = \{[y,z]|rs\}, [x,z] \; \mathsf{nin} \; rq, [y,z] \; \mathsf{nin} \; rs, \mathsf{comp}(rr,s,rq)\}$$

$$\frac{q \in \mathcal{V} \qquad r = \{[x,y]|rr\} \qquad \neg\mathsf{empty}(s) \qquad s \notin \mathcal{V}}{\{\mathsf{comp}(r,s,q)\} \to \{s = \{[y,z]|rs\},}$$
$$q = \{[x,z]|rq\}, [x,y] \; \mathsf{nin} \; rr, [y,z] \; \mathsf{nin} \; rs, \mathsf{comp}(rr,s,rq)\}$$
$$\text{or}$$
$$\{\mathsf{comp}(r,s,q)\} \to \{\mathsf{dom}(s,ds), y \; \mathsf{nin} \; ds, [x,y] \; \mathsf{nin} \; rr,$$
$$\mathsf{comp}(rr,s,q)\}$$

Fig. 2. Rewrite rules for comp.

$(n, m \geq 0)$ represents the changes in the constraint store caused by the rule application.

Some rewrite rules for dealing with single $\mathcal{PF}$-constraints are shown in Figures 1 and 2; all of them can be found in the online appendix (Appendix A). Rewrite rules for all other primitive constraints can be found elsewhere (Dovier et al. 2000; Dal Palù et al. 2003).

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 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. Notice that STEP returns

false whenever (at least) one of the procedures in it rewrites C to false. Moreover, STEP(false) returns false.

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

   procedure $SAT_{\mathcal{PF}}(C)$
      $C \leftarrow \mathsf{infer}(C)$
      repeat
         $C' \leftarrow C;$
         $C \leftarrow \mathsf{STEP}(C);$
      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 a 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.

Roughly speaking, the irreducible constraints are these because we are not able to rewrite them to *finite* conjunctions of primitive $(\mathcal{SET}, \mathcal{PF})$-constraints. In particular, solving the constraint $\mathsf{ran}(X, t)$, where t is a set term not denoting the empty set, would amount to solve the formula $\forall x(x \in X \Leftrightarrow \exists y, z(x = [y, z] \wedge z \in t))$ which is not expressible as a finite conjunction of primitive $(\mathcal{SET}, \mathcal{PF})$-constraints. Notice that, conversely, the case $\mathsf{dom}(X, t)$, where t is a set term (e.g. $\mathsf{dom}(X, \{1\})$), can be easily rewritten to a finite conjunction of primitive constraints since the cardinality of X is necessarily that of t; hence this constraint is not dealt with as irreducible.

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 Dovier et al. (2000)). Observe that, a constraint composed of only solved form literals is proved to be always satisfiable.

Example 2
Constraint rewriting.

- $\mathsf{dom}(\{[a, 1], [b, 2], [c, 1]\}, D)$ is rewritten to $D = \{a, b, c\}$
- $\mathsf{dom}(\{[a, 1]\}, \{b\})$ is rewritten to false
- $\mathsf{comp}(\{[1, b]\}, B, \{[1, a]\})$ is rewritten to $B = \{[b, a] | BR\} \wedge [b, a]$ nin $BR \wedge$ $\mathsf{pfun}(BR) \wedge \mathsf{dom}(BR, D) \wedge b$ nin $D \wedge \mathsf{set}(D)$
- $\mathsf{inters}(\{X\}, \{1\}, D) \wedge \mathsf{dom}(R, D) \wedge \mathsf{ran}(R, \emptyset)$ is rewritten to $D = \emptyset \wedge R = \emptyset \wedge X$ neq 1

- apply$(F, X, Y) \land$ dom$(F, D) \land X$ nin D is rewritten to false.

Note that with the implementation of dom and ran as user-defined $\{log\}$ predicates (see Cristiá et al. (2013)) the last goal 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, returning either false or 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, hence, it is correct.

Theorem 3 (*Equisatisfiability*)
Let C be a constraint, $C_1, \ldots, C_n$ be the constraints obtained from $SAT_{\mathcal{PF}}(C)$, σ be a valuation of C and $C_1 \lor \ldots \lor C_n$, 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 $\mathcal{A}_{\mathcal{PF}} \models \sigma(C_1 \lor \ldots \lor C_n)$.

If at least one of the constraints C_i returned by $SAT_{\mathcal{PF}}(C)$ contains *only* primitive $\mathcal{SET}$-constraints then, according to Dovier et al. (2000), 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 3
The following $(\mathcal{SET}, \mathcal{PF})$-constraint

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

is an irreducible constraint but it is clearly unsatisfiable (the only possible solution for un$(D, Y, Z) \land$ disj(D, Z) is $D = \emptyset$, but $D = \emptyset$ 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 primitive constraint pfun, of arity 2, with the following interpretation: $\mathsf{pfun}(r, n)$ holds 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 than 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.

¿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* over the pf-variable r. Note 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 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, it is straightforward to see that the rewriting rules for pfun/2 preserve the set of solutions of the input constraint. Thus we can immediately extend to pfun/2 constraints the results of Theorems 2 and 3.

Solving 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 an 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 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. Since $SAT_{\mathcal{PF}}$ is proved to preserve the set of solutions of C (cf. Theorem 3), then we can conclude that in this case C is satisfiable.

Hence, by properly exploiting pfun/2 constraints, we get a complete solver. This means that, once $k_1, \ldots, k_n$ are fixed, 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 4
The following constraints are rewritten to either false or to a solved form constraint, whereas they are left unchanged if no pf-domain is specified.

- $\mathsf{dom}(R, D) \wedge D \ \mathsf{neq} \ \emptyset \wedge \mathsf{un}(D, Y, Z) \wedge \mathsf{disj}(D, Z) \wedge \mathsf{pfun}(R, 5)$ is rewritten to false
- $\mathsf{ran}(X, \{1\}) \wedge \mathsf{un}(X, Y, Z) \wedge \mathsf{pfun}(X, 5)$ is rewritten to the solved form constraint (first solution): $X = \{[A, 1]\} \wedge Z = \{[A, 1]|Y\} \wedge \mathsf{set}(Y)$.

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 constraint

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

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.

Some of the inference rules used by $SAT'_{\mathcal{PF}}$ are shown in Figure 3; all of them can be found in the online appendix (Appendix A). 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.

Example 5
The following constraints are all proved to be unsatisfiable using $SAT'_{\mathcal{PF}}$ (see the applied rules in Figure 3):

$\mathsf{dom}(X, D1) \wedge \mathsf{dom}(X, D2) \wedge D1 \ \mathsf{neq} \ D2$	(rule (1))
$\mathsf{ran}(X, RX) \wedge RX \ \mathsf{neq} \ \emptyset \wedge \mathsf{disj}(X, Z) \wedge \mathsf{un}(X, Y, Z)$	(rule (2))
$\mathsf{dom}(X, DX) \wedge \mathsf{size}(X, N) \wedge \mathsf{size}(DX, M) \wedge N \ \mathsf{neq} \ M$	(rule (3))
$\mathsf{comp}(\{[a, 1]\}, Y, Z) \wedge \mathsf{dom}(Z, DZ) \wedge a \ \mathsf{nin} \ DZ \wedge Z \ \mathsf{neq} \ \emptyset$	(rule (4))
$\mathsf{un}(X, Y, Z) \wedge \mathsf{dom}(X, D) \wedge \mathsf{dom}(Y, D) \wedge \mathsf{dom}(Z, DZ) \wedge D \ \mathsf{neq} \ DZ$	(rule (5))

$$\frac{}{\{\mathsf{dom}(r,a),\mathsf{dom}(r,b)\} \to \{\mathsf{dom}(r,a), a=b\}} \tag{1}$$

$$\frac{a \in \mathcal{V}}{\{\mathsf{ran}(r,a), r \ \mathsf{neq}\ \emptyset\} \to \{\mathsf{ran}(r,a), r \ \mathsf{neq}\ \emptyset, a \ \mathsf{neq}\ \emptyset\}} \tag{2}$$

$$\frac{}{\{\mathsf{dom}(r,a)\} \to \{\mathsf{dom}(r,a), \mathsf{size}(r,n), \mathsf{size}(a,n)\}} \tag{3}$$

$$\frac{}{\{\mathsf{comp}(r,s,q)\} \to \begin{array}{l}\{\mathsf{comp}(r,s,q),\\ \mathsf{dom}(q,a), \mathsf{dom}(r,b), \mathsf{subset}(a,b)\}\end{array}} \tag{4}$$

$$\frac{}{\{\mathsf{un}(r,s,q), \mathsf{pfun}(q)\} \to \begin{array}{l}\{\mathsf{un}(r,s,q), \mathsf{pfun}(q),\\ \mathsf{dom}(r,dr), \mathsf{dom}(s,ds), \mathsf{dom}(q,dq),\\ \mathsf{un}(dr,ds,dq)\}\end{array}} \tag{5}$$

Fig. 3. Some inference rules.

The same constraints of Example 5 but using $SAT_{\mathcal{PF}}$, that is without applying any inference rule, are simply treated as irreducible. On the other hand, adding constraints 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 of the extended solver $SAT'_{\mathcal{PF}}$ comes from soundness 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. 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 unless pfun/2 is used for all pf-variables. 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 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. However, finding a collection of inference rules that guarantees 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 Empirical Assessment

In this section we present how the new version of $\{log\}$ (i.e. 4.8.2-2) improves its efficiency and effectiveness when solving formulas including partial functions and their operators. To do so we have generated more than 2,000 goals, some of which include partial functions and the related operators. These goals have been used to evaluate $\{log\}$ 4.8.0 as a test case generator for Fastest, a model-based testing tool (Cristiá et al. 2013). Besides, these goals have been generated by Fastest from 10 different Z specifications, some of which are formalizations of real requirements and, in general, they cover a wide range of applications—totalizing around 3,000 lines of Z code. These goals not only include partial functions, but also sets (in particular intentional sets), integer and relational constraints. Thus, we consider that they are a representative sample.

In this assessment, we want to know: (i) how many satisfiable and unsatisfiable goals are found by $\{log\}$; (ii) how long it takes to process all the goals; (iii) how $\{log\}$ performs in each task compared with version 4.8.0 (which do not include partial functions as primitive constraints).

Experiments were run on a 4 core Intel Core^{TM} i5-2410M CPU at 2.30GHz with 4 Gb of main memory, running Linux Ubuntu 12.04 (Precise Pangolin) 32-bit with kernel 3.2.0-80-generic-pae. $\{log\}$ 4.8.0 and 4.8.2-2 over SWI-Prolog 6.6.6 for i386 were used during the experiments. A 10 seconds timeout was set as the maximum time that $\{log\}$ can spend to give an answer for a goal.

Table 1 displays the results of the experiments. The meaning of the columns is as follows: **Z Spec**, Z specification; **Goals**, number of goals processed during the experiment; **S**, number of satisfiable goals detected as satisfiable; **U**, number of goals detected as unsatisfiable; **A**, percentage of goals for which $\{log\}$ gives a meaningful answer (i.e. $\mathbf{A} = 100(\mathbf{S} + \mathbf{U})/\mathbf{Goals}$); **T**, time spent by $\{log\}$ during the entire execution.

As can be seen, $\{log\}$ 4.8.2-2 outperforms 4.8.0 in almost all sets of goals. In effect, in all sets but two (SWPDC and Sec. class) 4.8.2-2 gives more right answers and in less time than 4.8.0. Note that 4.8.2-2 hits 100% of right answers in 5 sets of goals while 4.8.0 does it only in 2. Also note the impressive time reduction in, for example, Launcher. Given that giving more right answers in less time is the best behavior, we can define QI, for quality index, as $QI = \lfloor 100 * \mathbf{A}/\mathbf{T} \rfloor$. Then, the higher the QI the better. $\{log\}$ 4.8.2-2 has higher or equal QI than 4.8.0 in all but one set of goals.

Table 1. *Summary of empirical assessment*

Z Spec	Goals	4.8.0				4.8.2-2			
		S	**U**	**A**	**T**	**S**	**U**	**A**	**T**
SWPDC	196	97	26	63%	1,238	99	26	64%	1,402
Plavis	232	151	36	81%	510	151	33	79%	510
Scheduler	205	27	85	55%	945	38	161	97%	125
Sec. class	36	20	16	100%	11	20	14	94%	31
Bank (1)	100	23	39	62%	388	25	75	100%	28
Bank (3)	104	50	35	82%	211	52	49	97%	64
Lift	17	17	0	100%	6	17	0	100%	6
Launcher	1,206	0	1,093	91%	1,334	23	1,183	100%	370
Symb. table	27	11	10	78%	68	11	16	100%	9
Sensors	16	7	3	63%	54	8	8	100%	5
Totals	2,139	403	1,343	–	4,769	444	1,565	–	2,552

In summary, the experimental results show that adding constraints for partial functions as $\{log\}$'s primitive constraints greatly improves its efficiency and effectiveness as a constraint solver for a very general theory of sets.

8 Conclusions

In this paper we have shown how to integrate partial functions as first-class citizens into the CLP 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 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: (i) 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 $\mathsf{ran}(X, \{\dots\})$)) ($ii$) 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

ABRIAL, J.-R. 1996. *The B-book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA.

CRISTIÁ, M., ROSSI, G., AND FRYDMAN, C. S. 2013. {log} as a test case generator for the Test Template Framework. In *SEFM*, R. M. Hierons, M. G. Merayo, and M. Bravetti, Eds. Lecture Notes in Computer Science, vol. 8137. Springer, 229–243.

DAL PALÙ, A., DOVIER, A., PONTELLI, E., AND ROSSI, G. 2003. Integrating finite domain constraints and CLP with sets. In *PPDP*. ACM, 219–229.

DEVILLE, Y., DOOMS, G., ZAMPELLI, S., AND DUPONT, P. 2005. CP(graph+map) for approximate graph matching. In *1st International Workshop on Constraint Programming Beyond Finite Integer Domains.* 31–47.

DOVIER, A., OMODEO, E. G., PONTELLI, E., AND ROSSI, G. 1996. A language for programming in logic with finite sets. *J. Log. Program. 28,* 1, 1–44.

DOVIER, A., PIAZZA, C., PONTELLI, E., AND ROSSI, G. 2000. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst. 22,* 5, 861–931.

GERVET, C. 1997. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints 1,* 3, 191–244.

GERVET, C. 2006. *Handbook of Constraint Programming.* Elsevier, Chapter Constraints over Structured Domains, 605–638.

JACKSON, D. 2003. Alloy: A logical modelling language. In *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, D. Bert, J. P. Bowen, S. King, and M. A. Waldén, Eds. Lecture Notes in Computer Science, vol. 2651. Springer, 1.

ROSSI, G. 2008. {*log*}. `http://people.math.unipr.it/gianfranco.rossi/setlog.Home.html`. last access: May 2015.

SAALTINK, M. 1997. The Z/EVES mathematical toolkit version 2.2 for Z/EVES version 1.5. Tech. rep., ORA Canada.

SCHWARTZ, J. T., DEWAR, R. B. K., DUBINSKY, E., AND SCHONBERG, E. 1986. *Programming with Sets - An Introduction to SETL.* Texts and Monographs in Computer Science. Springer.

SPIVEY, J. M. 1992. *The Z notation: a reference manual.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.