Half Reification and Flattening

Thibaut Feydy[1], Zoltan Somogyi[1], and Peter J. Stuckey[1]

National ICT Australia and the University of Melbourne, Victoria, Australia
{tfeydy,zs,pjs}@csse.unimelb.edu.au

Abstract. Usually propagation-based constraint solvers construct a constraint network as a conjunction of constraints. They provide propagators for each form of constraint c. In order to increase expressiveness, systems also usually provide propagators for reified forms of constraints. A reified constraint $b \leftrightarrow c$ associates a truth value b with a constraint c. With reified propagators, systems can express complex combinations of constraints using disjunction, implication and negation by flattening. In this paper we argue that reified constraints should be replaced by half-reified constraints of the form $b \rightarrow c$. Half-reified constraints do not impose any extra burden on the implementers of propagators compared to unreified constraints, they can implement reified propagators without loss of propagation strength (assuming c is negatable), they extend automatically to global constraints, they simplify the handling of partial functions, and can allow flattening to give better propagation behavior.

1 Introduction

Constraint programming propagation solvers solve constraint satisfaction problems of the form $\exists V. \wedge_{c \in C} c$, that is an existentially quantified conjunction of primitive constraints c. But constraint programming modeling languages such as OPL [1], Zinc/MiniZinc [2, 3] and Essence [4] allow much more expressive problems to be formulated. Modeling languages map the more expressive formulations to existentially quantified conjunction through a combination of loop unrolling, and flattening using reification.

Example 1. Consider the following "complex constraint" written in Zinc syntax

```
constraint i <= 4 -> a[i] * x >= 6;
```

which requires that if $i \leq 4$ then the value in the i^{th} position of array a multiplied by x must be at least 6. This becomes the following existentially quantified conjunction through flattening and reification:

```
constraint b1 <-> i <= 4;   % b1 holds iff i <= 4
constraint element(i,a,t1); % t1 is the ith element of a
constraint mult(t1,x,t2);   % t2 is t1 * x
constraint b2 <-> t2 >= 6;  % b2 holds iff t2 >= 6
constraint b1 -> b2         % b1 implies b2
```

The complex logic (implication) is encoded by "reifying" the arguments and in effect naming their truth value using new Boolean variables $b1$ and $b2$. The term structure is encoded by "flattening" the terms and converting the functions to relations, introducing the new integer variables $t1$ and $t2$. Note that the newly introduced variables are existentially quantified. $\square$

The translation given in the above example is well understood, but potentially flawed, for three reasons. The first is that the flattening may not give the intuitive meaning when functions are partial.

Example 2. Suppose the array a has index set 1..5, but i takes the value 7. The constraint `element`$(i, a, t1)$ will fail and no solution will be found. Intuitively if $i = 7$ the constraint should be trivially true. $\square$

The simple flattening used above treats partial functions in the following manner. Application of a partial function to a value for which it is not defined gives value $\perp$, and this $\perp$ function percolates up through every expression to the top level conjunction, making the model unsatisfiable. For the example $(t1 \equiv)\ a[7] = \perp$, $(t2 \equiv) \perp \times x = \perp$, $(b2 \equiv) \perp \geq 6 = \perp$, $(b1 \equiv)\ 7 \leq 4 = \mathit{false}$, $\mathit{false} \rightarrow \perp = \perp$. This is known as the *strict* semantics [5] for modeling languages.

The usual choice for modeling partial functions in modeling languages is the *relational* semantics [5]. In the relational semantics the value $\perp$ percolates up through the term until it reaches a Boolean subterm where it becomes *false*. For the example $(t1 \equiv)\ a[7] = \perp$, $(t2 \equiv) \perp \times x = \perp$, $(b2 \equiv) \perp \geq 6 = \mathit{false}$, $(b1 \equiv)\ 7 \leq 4 = \mathit{false}$, $\mathit{false} \rightarrow \mathit{false} = \mathit{true}$. But in order to implement the relational semantics, the translation of the original complex constraint needs to be far more complex.

Example 3. The tool `mzn2fzn` unrolls, flattens, and reifies MiniZinc models implementing the relational semantics. Assuming i takes values in the set 1..8, and a has an index set 1..5, its translation of the constraint in Example 1 is

```
constraint b1 <-> i <= 4;   % b1 holds iff i <= 4
constraint element(t3,a,t1);% t1 is the t3'th element of a
constraint mult(t1,x,t2);   % t2 is t1 * x
constraint b2 <-> t2 >= 6;  % b2 holds iff t2 >= 6
constraint t3 in 1..5       % t3 in index set of a
constraint b3 <-> i = t3;   % b3 holds iff i = t3
constraint b3 <-> i <= 5;   % b3 holds iff i in index set of a
constraint b4 <-> b2 /\ b3  % b4 holds iff b2 and b3 hold
constraint b1 -> b4         % b1 implies b4
```

The translation forces the partial function application `element` to be "safe" since $t3$ is constrained to only take values in the index set of a. The reified constraints defining $b3$ force $t3$ to equal i iff i takes a value in the index set of a. $\square$

A second weakness of reification, independent of the problems with partial functions, is that each reified version of a constraint requires further implementation to create, and indeed most solvers do not provide any reified versions of their global constraints.

Example 4. Consider the complex constraint

```
constraint i <= 4 -> alldifferent([i,x-i,x]);
```

The usual flattened form would be

```
constraint b1 <-> i <= 4;   % b1 holds iff i <= 4
constraint minus(x,i,t1);   % t1 = x - i
constraint b2 <-> alldifferent([i,t1,x]);
constraint b1 -> b2         % b1 implies b2
```

but no solver we are aware of implements the third primitive constraint.[1] □

Reified global constraints are not implemented because a reified constraint $b \leftrightarrow c$ must also implement a propagator for $\neg c$ (in the case that $b = \mathit{false}$). While for some global constraints, e.g. `alldifferent`, this may be reasonable to implement, for most, such as `cumulative`, the task seems to be very difficult.

A third weakness of the full reification is that it may keep track of more information than is required. In a typical finite domain solver, the first reified constraint $b1 \leftrightarrow i \leq 4$ will wake up whenever the lower bound of i changes in order to check whether it should set $b1$ to *false*. But setting $b1$ to *false* will *never* cause any further propagation. There is no reason to check this.

Flattening with half-reification is an approach to mapping complex constraints to existentially quantified conjunctions that improves upon all these weaknesses of flattening with full reification.

- Flattening with half reification can naturally produce the relational semantics when flattening partial functions in positive contexts.
- Half reified constraints add no burden to the solver writer; if they have a propagator for constraint c then they can straightforwardly construct a half reified propagator for $b \rightarrow c$.
- Half reified constraints $b \rightarrow c$ can implement fully reified constraints without any loss of propagation strength (assuming reified constraints are negatable).
- Flattening with half reification can produce more efficient propagation when flattening complex constraints.

Our conclusion is that propagation solvers *only* need to provide half reified version of all constraints. This does not burden the solver writer at all, yet it provides more efficient translation of models, and more expressiveness in using global constraints.

2 Propagation Based Constraint Solving

We consider a typed set of variables $\mathcal{V} = \mathcal{V}_I \cup \mathcal{V}_B$ made up of *integer* variables, $\mathcal{V}_I$, and *Boolean* variables, $\mathcal{V}_b$. We use lower case letters such as x and y for integer variables and letters such as b for Booleans. A *domain* D is a complete mapping from $\mathcal{V}$ to finite sets of integers (for the variables in $\mathcal{V}_I$) and to subsets of $\{\mathit{true}, \mathit{false}\}$ (for the variables in $\mathcal{V}_b$). We can understand a domain D as a

[1] Although there are versions of soft `alldifferent`, they do not define this form.

formula $\wedge_{v \in \mathcal{V}}(v \in D(v))$ stating for each variable v that its value is in its domain. A *false domain* D is a domain where $\exists v \in \mathcal{V}. D(v) = \emptyset$, and corresponds to an unsatisfiable formula.

Let D_1 and D_2 be domains and $V \subseteq \mathcal{V}$. We say that D_1 is *stronger* than D_2, written $D_1 \sqsubseteq D_2$, if $D_1(v) \subseteq D_2(v)$ for all $v \in \mathcal{V}$ and that D_1 and D_2 are *equivalent modulo* V, written $D_1 =_V D_2$, if $D_1(v) = D_2(v)$ for all $v \in V$. The *intersection* of D_1 and D_2, denoted $D_1 \sqcap D_2$, is defined by the domain $D_1(v) \cap D_2(v)$ for all $v \in \mathcal{V}$. We assume an *initial domain* D_{init} such that all domains D that occur will be stronger i.e. $D \sqsubseteq D_{init}$.

A *valuation* θ is a mapping of integer and Boolean variables to correspondingly typed values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n, b_1 \mapsto tf_1, \ldots, b_m \mapsto tf_m\}$. We extend the valuation θ to map expressions or constraints involving the variables in the natural way. Let *vars* be the function that returns the set of variables appearing in an expression, constraint or valuation. In an abuse of notation, we define a valuation θ to be an element of a domain D, written $\theta \in D$, if $\theta(v) \in D(v)$ for all $v \in vars(\theta)$.

A constraint is a restriction placed on the allowable values for a set of variables. We define the *solutions* of a constraint c to be the set of valuations θ that make that constraint true, i.e. $solns(c) = \{\theta \mid (vars(\theta) = vars(c)) \wedge (\models \theta(c))\}$

We associate with every constraint c a *propagator* f_c. A propagator f_c is a monotonically decreasing function on domains such that for all domains $D \sqsubseteq D_{init}$: $f_c(D) \sqsubseteq D$ and $\{\theta \in D \mid \theta \in solns(c)\} = \{\theta \in f_c(D) \mid \theta \in solns(c)\}$. This is a weak restriction since, for example, the identity mapping is a propagator for any constraint.

A domain D is *domain consistent* for constraint c if $D(v) = \{\theta(v) \mid \theta \in solns(c) \wedge \theta \in D\}$, for all $v \in vars(c)$. A domain D is *bounds(Z) consistent* for constraint c over variables $v_1, \ldots v_n$ if for each $i \in \{1, \ldots, n\}$ there exists $\theta \in solns(c) \cap D$ s.t. $\theta(v_i) = \min D(v_i)$ and $\min D(v_j) \leq \theta(v_j) \leq \max D(v_j), 1 \leq j \neq i \leq n$, and similarly exists $\theta \in solns(c) \cap D$ s.t. $\theta(v_i) = \max D(v_i)$ and $\min D(v_j) \leq \theta(v_j) \leq \max D(v_j), 1 \leq j \neq i \leq n$. For Boolean variables v we assume *false* $<$ *true*. A domain D is *bounds(R) consistent* for constraint c if the same conditions as for bounds(Z) consistency hold except $\theta \in solns(c')$ where c' is the *real relaxation* of c. Note that we assume Booleans can only take Boolean values in the real relaxation.

Note that for the pure Boolean constraints domain, bounds(Z) and bounds(R) consistency coincide.

A propagator f_c is X-consistent if $f(D)$ is always X consistent for c, where X could be domain, bounds(Z) or bounds(R).

A *propagation solver* for a set of propagators F and current domain D, $solv(F, D)$, repeatedly applies all the propagators in F starting from domain D until there is no further change in the resulting domain. $solv(F, D)$ is the weakest domain $D' \sqsubseteq D$ which is a fixpoint (i.e. $f(D') = D'$) for all $f \in F$. In other words, $solv(F, D)$ returns a new domain defined by $solv(F, D) = \text{gfp}(\lambda d.iter(F, d))(D)$ where $iter(F, D) = \bigsqcap_{f \in F} f(D)$, where gfp denotes the greatest fixpoint w.r.t $\sqsubseteq$ lifted to functions.

2.1 A Language of Constraints

For simplicity of presentation we restrict ourselves in this paper to the following simple grammar of constraints (a subset of MiniZinc), in which the cons nonterminal defines constraints, and the term nonterminal defines integer terms:

```
cons ⟶ true | false | bvar| term relop term
     ⟶ not cons | cons /\ cons | cons \/ cons | cons -> cons | cons <-> cons
     ⟶ pred(term_1, ..., term_n)
term ⟶ int | ivar | term arithop term | array[term] | bool2int( cons )
```

The grammar uses the symbols bvar for Boolean variables, relop for relational operators { ==, <=, <, !=, >=, > }, pred for names of builtin predicate constraints, int for integer constants, ivar for integer variables, arithop for arithmetic operators { +, -, *, div } and array for array constants. The main missing things are looping constructs, long linear and Boolean constraints, and local variables.

We assume each integer variable x is separately declared with a finite initial set of possible values $D_{init}(x)$. We assume each array constant is separately declared as a mapping $\{i \mapsto d \mid i \in idx(a)\}$ where its index set $idx(a)$ is a finite integer range. Given these initial declarations, we can determine the set of possible values of any term t in the language as $\{\theta(t) \mid \theta \in D_{init}\}$. Note also while it may be prohibitive to determine the set of possible values for any term t, we can efficiently determine a superset of these values by building a superset for each subterm bottom up using approximation.

Given a cons term defining the constraints of the model we can split its cons subterms as occurring in kinds of places: positive contexts, negative contexts, and mixed contexts. A Boolean subterm t of constraint c, written $c[t]$, is in a *positive context* iff for any solution θ of c then θ is also a solution of $c[true]$, that is c with subterm t replaced by *true*. Similarly, a subterm t of constraint c is in a *negative context* iff for any solution θ of c then θ is also a solution of $c[false]$. The remaining Boolean subterms of c are in mixed contexts.

Example 5. Consider the constraint expression c

```
constraint i <= 4 -> x + bool2int(b) = 5;
```

then $i \leq 4$ is in a negative context, $x + \texttt{bool2int}(b) = 5$ is in a positive context, and b is in a mixed context. If the last equality were $x + \texttt{bool2int}(b) \geq 5$ then b would be in a positive context. □

One can classify most contexts as positive or negative using a simple top-down analysis of the form of the expression. The remaining contexts can be considered mixed without compromising the correctness of the rest of the paper.

Our small language contains two partial functions: div returns $\perp$ if the divisor is zero, while $a[i]$ returns $\perp$ if the value of i is outside the domain of a. We can categorize the *safe* terms and constraints of the language, as those where no $\perp$ can ever arise in any subterm. A term or constraint is *safe* if all its arguments are safe, and either the term is not a division or array access, or it is a division term t_1 div t_2 and the set of possible values of t_2 does not include 0, or it is an array access term $a[t]$ and the set of possible values of t are included in $idx(a)$.

3 Flattening with Full Reification

Since the constraint solver only deals with a flat conjunction of constraints, modeling languages that support more complex constraint forms need to *flatten* them into a form acceptable to the solver. The usual method for flattening complex formula of constraints is full reification. Given a constraint c the *full reified form* for c is $b \leftrightarrow c$, where $b \notin vars(c)$ is a Boolean variable naming the satisfied state of the constraint c.

The pseudo-code for flatc(b,c) flattens a constraint expression c to be equal to b, returning a set of constraints implementing $b \leftrightarrow c$. We flatten a whole model c using flatc($true, c$). In the pseudo-code the expressions **new** b and **new** v create a new Boolean and integer variable respectively.

The code assumes there are reified versions of the basic relational constraints r available, as well as reified versions of the Boolean connectives. Flattening of arbitrary constraint predicates aborts if not at the top level of conjunction. The code handles unsafe terms by capturing them when they first arrive at a Boolean context using safen.

```
flatc($b$,$c$)
  switch $c$
  case true: return $\{b\}$
  case false: return $\{\neg b\}$
  case $b'$ (bvar): return $\{b \leftrightarrow b'\}$
  case $t_1$ $r$ $t_2$ (relop): return safen($b$, flatt(new $i_1, t_1$) $\cup$ flatt(new $i_2, t_2$)) $\cup$ $\{b \leftrightarrow i_1 \ r \ i_2\}$
  case not $c_1$: return flatc(new $b_1, c_1$) $\cup$ $\{b \leftrightarrow \neg b_1\}$
  case $c_1$ /\ $c_2$: if ($b \equiv true$) return flatc($true, c_1$) $\cup$ flatc($true, c_2$)
                 else return flatc(new $b_1, c_1$) $\cup$ flatc(new $b_2, c_2$) $\cup$ $\{b \leftrightarrow (b_1 \wedge b_2)\}$
  case $c_1$ \/ $c_2$: return flatc(new $b_1, c_1$) $\cup$ flatc(new $b_2, c_2$) $\cup$ $\{b \leftrightarrow (b_1 \vee b_2)\}$
  case $c_1$ -> $c_2$: return flatc(new $b_1, c_1$) $\cup$ flatc(new $b_2, c_2$) $\cup$ $\{b \leftrightarrow (b_1 \rightarrow b_2)\}$
  case $c_1$ <-> $c_2$: return flatc(new $b_1, c_1$) $\cup$ flatc(new $b_2, c_2$) $\cup$ $\{b \leftrightarrow (b_1 \leftrightarrow b_2)\}$
  case $p$ $(t_1, \ldots, t_n)$ (pred):
    if ($b \equiv true$) return safen($b$, $\cup_{j=1}^{n}$flatt(new $v_j, t_j$)) $\cup$ $\{p(v_1, \ldots, v_n)\}$
    else abort
```

The code flatt(v, t) flattens an integer term t, creating constraints that equate the term with variable v. It creates new variables to store the values of subterms, replaces integer operations by their relational versions, and array lookups by `element`.

```
flatt($v$,$t$)
  switch $t$
  case $i$ (int): return $\{v = i\}$
  case $v'$ (ivar): return $\{v = v'\}$
  case $t_1$ $a$ $t_2$ (arithop): return flatt(new $v_1, t_1$) $\cup$ flatt(new $v_2, t_2$) $\cup$ $\{a(v_1, v_2, v)\}$
  case $a$ [ $t_1$ ]: return flatt(new $v_1, t_1$) $\cup$ $\{$element$(v_1, a, v)\}$
  case bool2int( $c_1$ ): return flatc(new $b_1, c_1$) $\cup$ $\{$bool2int$(b_1, v)\}$)
```

The procedure safen(b, C) enforces the relational semantics for unsafe expressions, by ensuring that the unsafe relational versions of partial functions are made safe. Note that to implement the *strict semantics* as opposed to the relational semantics we just need to define $\mathsf{safen}(b, C) = C$. If $b \equiv \mathit{true}$ then the relational semantics and the strict semantics coincide, so nothing needs to be done. The same is true if the set of constraints C is safe. For $div(x, y, z)$, the translation introduces a new variable y' which cannot be 0, and equates it to y if $y \neq 0$. The constraint $div(x, y', z)$ never reflects a partial function application. The new variable b' captures whether the partial function application returns a non $\bot$ value. For $\texttt{element}(v, a, x)$, it introduces a new variable v' which only takes values in $idx(a)$ and forces it to equal v if $v \in idx(a)$. A partial function application forces $b = \mathit{false}$ since it is the conjunction of the new variables b'. The %HALF% comments will be explained later.

```
safen($b$,$C$)
  if ($b \equiv true$) return $C$
  if ($C$ is a set of safe constraints) return $C$
  $B := \emptyset$; $S := \emptyset$
  foreach $c \in C$
    if ($c \equiv div(x, y, z)$ and $y$ can take value 0)
      $B := B \cup \{\mathbf{new}\ b'\}$
      $S := S \cup \{\mathbf{new}\ y' \neq 0, b' \leftrightarrow y \neq 0, b' \leftrightarrow y = y', div(x, y', z)\}$
      %HALF% $S := S \cup \{b' \leftrightarrow y \neq 0, b' \rightarrow div(x, y, z)\}$
    else if $c \equiv \texttt{element}(v, a, x)$ and $v$ can take a value outside the domain of $a$)
      $B := B \cup \{\mathbf{new}\ b'\}$
      $S := S \cup \{\mathbf{new}\ v' \in idx(a), b' \leftrightarrow v \in idx(a), b' \leftrightarrow v = v', \texttt{element}(v', a, x)\}$
      %HALF% $S := S \cup \{b' \leftrightarrow v \in idx(a), b' \rightarrow \texttt{element}(v, a, x)\}$
    else $S := S \cup \{c\}$
    return $S \cup \{b \leftrightarrow \wedge_{b' \in B} b'\}$)
```

The flattening algorithms above can produce suboptimal results in special cases, such as input with common subexpressions. Our implementation avoids generating renamed-apart copies of already-generated constraints, but for simplicity of presentation, we omit the algorithms we use to do this.

4 Half Reification

Given a constraint c, the *half-reified version* of c is a constraint of the form $b \rightarrow c$ where $b \notin vars(c)$ is a Boolean variable.

We can construct a propagator $f_{b \rightarrow c}$ for the half-reified version of c, $b \rightarrow c$, using the propagator f_c for c.

$$
\begin{array}{ll}
f_{b\rightarrow c}(D)(b) = \{\mathit{false}\} \cap D(b) & \text{if } f_c(D) \text{ is a false domain} \\
f_{b\rightarrow c}(D)(b) = D(b) & \text{otherwise} \\
f_{b\rightarrow c}(D)(v) = D(v) & \text{if } v \not\equiv b \text{ and } \mathit{false} \in D(b) \\
f_{b\rightarrow c}(D)(v) = f_c(D)(v) & \text{if } v \not\equiv b \text{ and } \mathit{false} \notin D(b)
\end{array}
$$

In practice most propagator implementations for c first check whether c is satisfiable, before continuing to propagate. For example, $\sum_i a_i x_i \leq a_0$ determines

$L = \sum_i min_D(a_i x_i) - a_0$ and fails if $L > 0$ before propagating; Regin's domain propagator for `alldifferent`$([x_1, \ldots, x_n])$ determines a maximum matching between variables and values first, if this is not of size n it fails before propagating; the timetable `cumulative` constraint determines a profile of necessary resource usage, and fails if this breaks the resource limit, before considering propagation. We can implement the propagator for $f_{b \to c}$ by only performing the checking part until $D(b) = \{true\}$.

Half reification naturally encodes the relational semantics for partial function applications in positive contexts. We associate a Boolean variable b with each Boolean term in an expression, and we ensure that all unsafe constraints are half-reified using the variable of the nearest enclosing Boolean term.

Example 6. Consider flattening of the constraint of Example 1. First we will convert it to an equivalent expression with only positive contexts

```
i > 4 \/ a[i] * x >= 6
```

There are three Boolean terms: the entire constraint, $i > 4$ and $a[i] \times x \geq 6$, which we name b_0, b_1 and b_2 respectively. The flattened form using half reification is

```
constraint b1 -> i > 4;
constraint b2 -> element(i,a,t1);
constraint mult(t1,x,t2);
constraint b2 -> t2 >= 6;
constraint b1 \/ b2;
```

The unsafe `element` constraint is half reified with the name of its nearest enclosing Boolean term. Note that if $i = 7$ then the second constraint makes $b2 = \mathit{false}$. Given this, the final constraint requires $b1 = \mathit{true}$, which in turn requires $i > 4$. Since this holds, the whole constraint is *true* with no restrictions on x. □

Half reification can handle more constraint terms than full reification if we assume that each global constraint predicate p is available in half-reified form. Recall that this places no new burden on the solver implementer.

Example 7. Consider the constraint of Example 4. Half reification results in

```
constraint b1 -> i > 4;
constraint minus(i,x,t1);   % t1 = i - x
constraint b2 -> alldifferent([i,t1,x]);
constraint b1 \/ b2         % b1 or b2
```

We can easily modify any existing propagator for `alldifferent` to support the half-reified form, hence this model is executable by our constraint solver. □

Half reification can lead to more efficient constraint solving, since it does not propagate unnecessarily.

Example 8. Consider the task decomposition of a `cumulative` constraint (see e.g. [6]) which includes constraints of the form

```
constraint sum(i in Tasks where i != j)
  (bool2int(s[i] <= s[j] /\ s[i]+d[i] > s[j]) * r[i]) <= L - r[j];
```

which requires that at the start time $s[j]$ of task j, the sum of resources r used by it and by other tasks executing at the same time is less than the limit L. Flattening with full reification produces constraints like this:

```
constraint b1[i] <-> s[i] <= s[j];
constraint plus(s[i],d[i],e[i]);     % e[i] = s[i] + d[i]
constraint b2[i] <-> e[i] > s[j];
constraint b3[i] <-> b1[i] /\ b2[i];
constraint bool2int(b3[i], a[i]);    % a[i] = bool2int(b3[i])
constraint sum(i in Tasks where i != j)( a[i] * r[i] ) <= L - r[j];
```

Whenever the start time of task i is constrained so that it does not overlap time $s[j]$, then $b3[i]$ is fixed to *false* and $a[i]$ to 0, and the long linear sum is awoken. But this is useless, since it cannot cause failure. The Boolean expression appears in a negative context, and half-reification produces

```
constraint b1[i] -> s[i] > s[j];
constraint plus(s[i],d[i],e[i]);     % e[i] = s[i] + d[i]
constraint b2[i] -> e[i] <= s[j];
constraint b3[i] -> b1[i] \/ b2[i];
constraint b4[i] <-> not b3[i];
constraint bool2int(b4[i], a[i]);    % a[i] = bool2int(b4[i])
constraint sum(i in Tasks where i != j)( a[i] * r[i] ) <= L - r[j];
```

which may seem to be more expensive since there are additional variables (the $b4[i]$), but since both $b4[i]$ and $a[i]$ are implemented by views [7], there is no additional runtime overhead. This decomposition will only wake the linear constraint when some task i is guaranteed to overlap time $s[j]$. □

Half reification can cause propagators to wake up less frequently, since variables that are fixed to *true* by full reification will never be fixed by half reification. This is advantageous, but a corresponding disadvantage is that variables that are fixed can allow the simplification of the propagator, and hence make its propagation faster. We can reduce this disadvantage by fully reifying Boolean connectives (which have low overhead) where possible in the half reification.

Flattening with Half Reification The procedure $\mathsf{halfc}(b, c)$ defined below returns a set of constraints implementing the half-reification $b \rightarrow c$. We flatten a whole model c using $\mathsf{halfc}(\mathit{true}, c)$. The half-reification flattening transformation uses half reification whenever it is in a positive context. If it encounters a constraint c_1 in a negative context, it negates the constraint if it is safe, thus creating a new positive context. If this is not possible, it defaults to the usual flattening approach using full reification. Note how for conjunction it does not need to introduce a new Boolean variable. Negating a constraint expression is done one operator at a time, and is defined in the obvious way. For example, negating $t_1 < t_2$ yields $t_1 >= t_2$, and negating c_1 `/\` c_2 yields `not` c_1 `\/` `not` c_2. Any negations on subexpressions will be processed by recursive invocations of the algorithm.

```
halfc($b$,$c$)
  switch $c$
  case true: return $\{\}$
  case false: return $\{\neg b\}$
  case $b'$ (bvar): return $\{b \rightarrow b'\}$
  case $t_1$ $r$ $t_2$ (relop): return halft($b$, new $i_1, t_1$) $\cup$ halft($b$, new $i_2, t_2$) $\cup$ $\{b \rightarrow i_1\ r\ i_2\}$
  case not $c_1$:
    if ($c_1$ is safe) return halfc($b$, negate($c_1$))
    else return flatc(new $b_1$, not $c_1$) $\cup$ $\{b \rightarrow b_1\}$
  case $c_1$ /\ $c_2$: return halfc($b, c_1$) $\cup$ halfc($b, c_2$)
  case $c_1$ \/ $c_2$: return halfc(new $b_1, c_1$) $\cup$ halfc(new $b_2, c_2$) $\cup$ $\{b \rightarrow (b_1 \vee b_2)\}$
  case $c_1$ -> $c_2$: return halfc($b$, not $c_1$ \/ $c_2$)
  case $c_1$ <-> $c_2$: return flatc(new $b_1, c_1$) $\cup$ flatc(new $b_2, c_2$) $\cup$ $\{b \rightarrow (b_1 \leftrightarrow b_2)\}$
  case $p$ $(t_1, \ldots, t_n)$ (pred): return $\cup_{j=1}^{n}$halft($b$, new $v_j, t_j$) $\cup$ $\{b \rightarrow p(v_1, \ldots, v_n)\}$
```

Half reification of terms returns a set of constraints that enforce $v = t$ if the term t is safe, and $b \rightarrow v = t$ otherwise. The most complex case is `bool2int`(c_1), which half-reifies c_1 if it is in a positive context, negates c_1 and half-reifies the result if c_1 is safe and in a negative context, and uses full flattening otherwise.

```
halft($b$,$v$,$t$)
  if ($t$ is safe) return flatt($v, t$)
  switch $t$
  case $i$ (int): return $\{b \rightarrow v = i\}$ % unreachable
  case $v'$ (ivar): return $\{b \rightarrow v = v'\}$ % unreachable
  case $t_1$ $a$ $t_2$ (arithop): halft($b$, new $v_1, t_1$) $\cup$ halft($b$, new $v_2, t_2$) $\cup$ $\{b \rightarrow a(v_1, v_2, v)\}$
  case $a$ [ $t_1$ ]: halft($b$, new $v_1, t_1$) $\cup$ $\{b \rightarrow$ element$(v_1, a, v)\}$
  case bool2int( $c_1$ ):
    if ($c_1$ is in a positive context) return halfc(new $b_1, c_1$) $\cup$ $\{b \rightarrow$ bool2int$(b_1, v)\}$)
    else if ($c_1$ is safe and in a negative context)
      halfc(new $b_1$, negate($c_1$)) $\cup$ $\{b \rightarrow$ bool2int(new $b_2, v$), $b_2 \leftrightarrow \neg b_1\}$)
    else return flatc(new $b_1, c_1$) $\cup$ $\{b \rightarrow$ bool2int$(b_1, v)\}$
```

Half reified constraints can also simplify the process of enforcing the relational semantics for full reification, since we have a half-reified version of the `div` and `element` constraints. The safen operation can be improved by replacing the lines above those labeled %HALF% by the lines labeled %HALF%.

Full Reification using Half Reification Usually splitting a propagator into two will reduce the propagation strength. We show that modeling $b \leftrightarrow c$ for primitive constraint c using half-reified propagators as $b \rightarrow c, b \leftrightarrow \neg b', b' \rightarrow \neg c$ does not do so.

To do so independent of propagation strength, we define the behaviour of the propagators of the half-reified forms in terms of the full reified propagator.

$$\begin{array}{ll}
f_{b\rightarrow c}(D)(b) = D(b) \cap (\{\mathit{false}\} \cup f_{b\leftrightarrow c}(D)(b)) & \\
f_{b\rightarrow c}(D)(v) = D(v) & \texttt{if } v \not\equiv b, \mathit{false} \in D(b) \\
f_{b\rightarrow c}(D)(v) = f_{b\leftrightarrow c}(D)(v) & \texttt{if } v \not\equiv b, \text{otherwise}
\end{array}$$

and

$$\begin{array}{ll}
f_{b' \to \neg c}(D)(b') = D(b') & \texttt{if } \{false\} \in f_{b \leftrightarrow c}(D)(b) \\
f_{b' \to \neg c}(D)(b') = D(b') \cap \{false\} & \text{otherwise} \\
f_{b' \to \neg c}(D)(v) = D(v) & \texttt{if } v \not\equiv b', false \in D(b') \\
f_{b' \to \neg c}(D)(v) = f_{b \leftrightarrow c}(D)(v) & \texttt{if } v \not\equiv b, \text{otherwise}
\end{array}$$

These definitions are not meant describe implementations, only to define how the half reified split versions of the propagator should act.

Theorem 1. $\forall D.\, solv(\{f_{b \leftrightarrow c}, f_{b' \leftrightarrow \neg b}\})(D) = solv(\{f_{b \to c}, f_{b' \to \neg c}, f_{b' \leftrightarrow \neg b}\}, D)$.

Proof. Let $V = vars(c)$. We only consider domains D at a fixpoint of the propagators $f_{b' \leftrightarrow \neg b}$, i.e. $D(b') = \{\neg d \mid d \in D(b)\}$. The proof is by cases of D. **(a)** Suppose $D(b) = \{true, false\}$. **(a-i)** If $\exists \theta \in solns(c)$ where $\theta \in D$ (c can still be true) and $\exists \theta' \in D$ where $vars(\theta) = V$ and $\theta \notin solns(c)$ (c can still be false). then $f_{b \leftrightarrow c}$ does not propagate. Clearly neither do either of $f_{b \to c}$ or $f_{b' \to \neg c}$. **(a-ii)** Suppose c cannot still be false ($\forall \theta \in D$ where $vars(\theta) = V$ then $\theta \in solns(c)$) then $f_{b \leftrightarrow c}(D)(b) = \{true\}$ and similarly $f_{b' \to \neg c}(D)(b') = \{false\}$ using the second case of its definition. The propagator for $f_{b' \leftrightarrow \neg b}$ will then make the domain of b equal $\{true\}$. There is no other propagation in any case. **(a-iii)** Suppose c cannot still be true ($\neg(\exists \theta \in D \cap solns(c))$) then $f_{b \leftrightarrow c}(D)(b) = \{false\}$ and $f_{b \to c}(D)(b) = \{false\}$ using the first case of its definition. Again there is no other propagation in any case except making the domain of b' equal $\{true\}$. **(b)** If $D(b) = \{true\}$ then clearly $f_{b \leftrightarrow c}$ and $f_{b \to c}$ act identically on variables in $vars(c)$. **(c)** If $D(b) = \{false\}$ then $D(b') = \{true\}$ and clearly $f_{b \leftrightarrow c}$ and $f_{b' \to \neg c}$ act identically on variables in $vars(c)$. □

The reason for the generality of the above theorem which defines the half-reified propagation strength in terms of the full reified propagator is that we can now show that for the usual notions of consistency, replacing a fully reified propagator leads to the same propagation. Note that the additional variable b' can be implemented as a view [7] in the solver and hence adds no overhead.

Corollary 1. *A domain (resp. bounds(Z), bounds(R)) consistent propagator for* $b \leftrightarrow c$ *propagates identically to domain (resp. bounds(Z), bounds(R)) consistent propagators for* $b \to c$, $b \leftrightarrow \neg b'$, $b' \to \neg c$. □

5 Experiments

We ran our experiments on a PC with a 2.80GHz Intel i7 Q860 CPU and 4Gb of memory. `www.cs.mu.oz.au/~pjs/half` has our experimental MiniZinc models and instances. The first experiment considers "QCP-max" problems which are defined as quasi-group completion problems where the `alldifferent` constraints are soft, and the aim is to satisfy as many of them as possible.

```
int: n; % size
array[1..n,1..n] of 0..n: s; % 0 = unfixed 1..n = fixed
array[1..n,1..n] of var 1..n: q; % qcp array;
```

Table 1. QCP-max problems: Average time (in seconds), number of solved instances (300s timeout).

	FD						FD + Explanations					
Instances	full		half		half-g		full		half		half-g	
`qcp-10` (x15)	20.1	14	20.0	14	20.0	14	0	15	0	15	0	15
`qcp-15` (x15)	204.6	6	179.9	7	174.1	7	2.5	15	1.5	15	1.0	15
`qcp-20` (x15)	300.0	0	289.0	1	286.0	1	115.7	11	127.5	10	114.2	10

```
constraint forall(i,j in 1..n where s[i,j] > 0)(q[i,j] = s[i,j]);
solve maximize
    sum(i in 1..n)(bool2int(alldifferent([q[i,j] | j in 1..n]))) +
    sum(j in 1..n)(bool2int(alldifferent([q[i,j] | i in 1..n])));

predicate alldifferent(array[int] of var int: x) =
    forall(i,j in index_set(x) where i < j)(x[i] != x[j]);
```

Note that this is not the same as requiring the maximum number of disequality constraints to be satisfied. The `alldifferent` constraints, while apparently in a mixed context, are actually in a positive context, since the maximization in fact is implemented by inequalities forcing at least some number to be true.

In Table 1 we compare three different resulting programs on QCP-max problems: full reification of the model above, using the `alldifferent` decomposition defined by the predicate shown (full), half reification of the model using the `alldifferent` decomposition (half), and half reification using a half-reified global `alldifferent` (half-g) implementing arc consistency (thus having the same propagation strength as the decomposition). We use standard QCP examples from the literature, and group them by size. We compare both a standard finite domain solver (FD) and a learning lazy clause generation solver (FD + Explanations). We use the same fixed search strategy of labeling the matrix in order left-to-right from highest to lowest value for all approaches to minimize differences in search.

Half reification of the decomposition is more efficient, principally because it introduces fewer Boolean variables, and the direct implementation of the half reified constraint is more efficient still. Note that learning can be drastically changed by the differences in the model and full solves one more instance in `qcp-20`, thus winning in that case. Apart from this instance, the half reified versions give an almost uniform improvement.

The second experiment shows how half reification can reduce the overhead of handling partial functions correctly. Consider the following model for determining a prize collecting path, a simplified form of prize collecting traveling salesman problem [8], where the aim is define an acyclic path from node 1 along weighted edges to collect the most weight. Not every node needs to be visited ($pos[i] = 0$).

```
int: n; % size
array[1..n,0..n] of int: p; % prize for edge (i,j) Note p[i,0] = 0
```

Table 2. Prize collecting paths: Average time (in seconds) and number of solved instances with a 300s timeout for various number of nodes.

	FD						FD + Explanations					
Nodes	full		half		extended		full		half		extended	
15-3-5 (x 10)	0.31	10	0.25	10	0.26	10	0.21	10	0.17	10	0.17	10
18-3-6 (x 10)	1.79	10	1.37	10	1.52	10	0.70	10	0.51	10	0.58	10
20-4-5 (x 10)	5.30	10	4.04	10	4.51	10	1.28	10	0.97	10	1.17	10
24-4-6 (x 10)	46.03	10	34.00	10	40.06	10	7.28	10	4.91	10	6.37	10
25-5-5 (x 10)	66.41	10	50.70	10	57.51	10	9.75	10	6.58	10	8.28	10
28-4-7 (x 10)	255.06	5	214.24	8	241.10	6	38.54	10	23.27	10	34.83	10
30-5-6 (x 10)	286.48	1	281.00	2	284.34	1	100.54	10	60.65	10	92.19	10
32-4-8 (x 10)	300.00	0	297.12	1	300.00	0	229.86	5	163.73	10	215.16	8

```
array[1..n] of var 0..n: next; % next posn in tour
array[1..n] of var 0..n: pos;  % posn on node i in path, 0 = notin
array[1..n] of var int: prize = [p[i,next[i]] | i in 1..n];
                            % prize for outgoing edge
constraint forall(i in 1..n)(
    (pos[i] = 0 <-> next[i] = 0) /\
    (next[i] > 1 -> pos[next[i]] = pos[i] + 1));
constraint alldifferent_except_0(next) /\ pos[1] = 1;
solve minimize sum(i in 1..n)(prize[i]);
```

It uses the global constraint `alldifferent_except_0` which constrains each element in the *next* array to be different or equal 0. The model has one unsafe array lookup $pos[next[i]]$. We compare using full reification (full) and half reification (half) to model this problem. Note that if we extend the *pos* array to have domain $0..n$ then the model becomes safe. We also compare against this model (extended). We use graphs with both positive and negative weights for the tests. The search strategy fixes the *next* variables in order of their maximum value. First we note that extended is slightly better than full because of the simpler translation, while half is substantially better than extended since most of the half reified `element` constraints become redundant. Learning increases the advantage because the half reified formulation focusses on propagation which leads to failure which creates more reusable nogoods.

In the final experiment we compare resource constrained project scheduling problems (RCPSP) where the `cumulative` constraint is defined by the task decomposition as in Example 8 above, using both full reification and half-reification. We use standard benchmark examples from PSPlib [9]. Table 3 compares RCPSP instances using full reification and half reification. We compare using J30 instances (J30) and instances due to Baptiste and Le Pape (BL). Each line in the table shows the average run time and number of solved instances. The search strategy tries to schedule the task with the earliest possible start time. We find a small and uniform speedup for half over full across the suites, which improves with learning, again because learning is not confused by propagations that do not lead to failure.

Table 3. RCPSP: Average time (in seconds) and number of solved instances with a 300s timeout.

	FD				FD + Explanations			
Instances	full		half		full		half	
BL (x 40)	277.2	5	269.3	5	17.1	39	15.4	39
J30 (x 480)	116.1	300	114.3	304	16.9	463	12.9	468

6 Related Work and Conclusion

Half reification on purely Boolean constraints is well understood, this is the same as detecting the *polarity* of a gate, and removing half of the clausal representation of the circuit (see e.g. [10]). The flattening of functions (partial or total) and the calculation of polarity for Booleans terms inside `bool2int` do not arise in pure Boolean constraints.

Half reified constraints have been used in constraint modeling but are typically not visible as primitive constraints to users, or produced through flattening. Indexicals [11] can be used to implement reified constraints by specifying how to propagate a constraint c, propagate its negation, check disentailment, and check entailment, and this is implemented in SICstus Prolog [12]. A half reified propagator simply omits entailment and propagating the negation. Half reified constraints appear in some constraint systems, for example SCIP [13] supports half-reified real linear constraints of the form $b \rightarrow \sum_i a_i x_i \leq a_0$ exactly because the negation of the linear constraint $\sum_i a_i x_i > a_0$ is not representable in an LP solver so full reification is not possible.

While flattening is the standard approach to handle complex formula involving constraints, there are a number of other approaches which propagate more strongly. Schulte [14] proposes a generic implementation of $b \leftrightarrow c$ propagating (the flattened form of) c in a separate constraint space which does not affect the original variables; entailment and disentailment of c fix the b variable appropriately, although when b is made *false* the implementation does not propagate $\neg c$. This can also be implemented using propagator groups [15]. Brand and Yap [16] define an approach to propagating complex constraint formulae called controlled propagation which ensures that propagators that cannot affect the satisfiability are not propagated. They note that for a formula without negation, they could omit half their control rules, corresponding to the case for half reification of a positive context. Jefferson *et al* [17] similarly define an approach to propagating positive constraint formulae by using watch literal technology to only wake propagators for reified constraints within the formula when they can affect the final result. They use half reified propagators, which they call the "reifyimplied" form of a constraint, in some of their constraint models, though they do not compare half reified models against full reified models. We can straightforwardly fit these stronger propagation approaches to parts of a constraint formula into the flattening approach by treating the whole formula as a predicate, and the implementation of the stronger propagation as its propagator.

We suggest that all finite domain constraint solvers should move to supporting half-reified versions of all constraints. This imposes no further burden on solver implementors, it allows more models to be solved, it can be used to implement full reification, and it can allow translation to more efficient models.

We are currently extending the translator from MiniZinc to FlatZinc, `mzn2fzn`, to also support half-reification. This means also extending FlatZinc to include half-reified versions of constraints.

Acknowledgments. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

References

1. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press (1999)
2. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P., Garcia de la Banda, M., Wallace, M.: The design of the Zinc modelling language. Constraints **13**(3) (2008) 229–267
3. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Procs. of CP2007. (2007) 529–543
4. Frisch, A.M., Grum, M., Jefferson, C., Hernandez, B.M., Miguel, I.: The design of ESSENCE: A constraint language for specifying combinatorial problems. In: Procs. of IJCAI-07. (2007)
5. Frisch, A., Stuckey, P.: The proper treatment of undefinedness in constraint languages. In: Procs. of CP2009. (2009) 367–382
6. Schutt, A., Feydy, T., Stuckey, P., Wallace, M.: Why cumulative decomposition is not as bad as it sounds. In: Procs. of CP2009. (2009) 746–761
7. Schulte, C., Tack, G.: Views iterators for generic constraint implementations. In: Procs. of CP2005. (2005) 817–821
8. Balas, E.: The prize collecting traveling salesman problem. Networks **19** (1989) 621–636
9. PSPlib: Project scheduling problem library http://129.187.106.231/psplib/.
10. Plaisted, D., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation **2** (1986) 293–304
11. Van Hentenryck, P., Saraswat, V., Deville, Y.: Constraint processins in cc(FD). Manuscript (1991)
12. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Procs. PLILP97. (1997) 191–206
13. Achterberg, T., Berthold, T., Koch, T., Wolter, K.: Constraint integer programming: A new approach to integrate CP and MIP. In: Proc. of CPAIOR 2008. (2008) 6–20
14. Schulte, C.: Programming deep concurrent constraint combinators. In: Procs. of PADL 2000. (2000) 215–229
15. Lagerkvist, M.Z., Schulte, C.: Propagator groups. In: Procs. of CP2009. Volume 5732 of LNCS., Springer (2009) 524–538
16. Brand, S., Yap, R.: Towards "Propagation = Logic + Control". In: Procs. ICLP'06: 22nd International Conference on Logic Programming. (2006) 102–106
17. Jefferson, C., Moore, N.C.A., Nightingale, P., Petrie, K.E.: Implementing logical connectives in constraint programming. Artif. Intell. **174**(16-17) (2010) 1407–1429