Promoting Constraints to First-Class Status

Tobias Müller

Programming System Lab, Universität des Saarlandes
Postfach 15 11 50, D-66041 Saarbrücken, Germany
tmueller@ps.uni-sb.de

Abstract. This paper proposes to promote constraints to first-class status. In contrast to constraint propagation, which performs inference on values of variables, first-class constraints allow reasoning about the constraints themselves. This lets the programmer access the current state of a constraint and control a constraint's behavior directly, thus making powerful new programming and inference techniques possible, as the combination of constraint propagation and rewriting constraints à la term rewriting. First-class constraints allow for *true* meta constraint programming. Promising applications in the field of combinatorial optimization include early unsatisfiability detection, constraint reformulation to improve propagation, garbage collection of redundant but not yet entailed constraints, and finding minimal inconsistent subsets of a given set of constraints for debugging immediately failing constraint programs.
We demonstrate the above-mentioned applications by means of examples. The experiments were done with Mozart Oz but can be easily ported to other constraint solvers.

Keywords: Constraint programming, first-class constraints, early failure detection, simplification and garbage collection of constraints, minimal sets of inconsistent constraints.

1 Introduction

This paper proposes to promote constraints to first-class status and presents three applications for combinatorial problems. In contrast to constraint propagation, which performs inference on values of variables, first-class constraints allow reasoning about the constraints themselves. This lets the programmer access the current state of a constraint and control a constraint's behavior directly, thus making powerful new programming and inference techniques possible, as the combination of constraint propagation and rewriting constraints à la term rewriting. Promising applications in the field of combinatorial optimization include early unsatisfiability detection, constraint reformulation to improve propagation, and garbage collection of redundant but not yet entailed constraints.

In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Proceedings of the First International Conference on Computational Logic – CL2000*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 429–447, London, UK, July 2000. Springer Verlag.

Commonly, a constraint that reflects its validity to a 0/1-variable is called a meta constraint. This notion is slightly misleading since this reflection does not allow for true meta programming in the sense of self-reasoning and self-modification. Hence Smolka coined the term *reified* constraints, which we use in this paper, instead of meta constraints (first used in [6]). First-class constraints are orthogonal to reified constraints and allow for *true* meta constraint programming. For example, one can obtain the name and the parameters of a first-class constraint and learn whether it is already entailed or not. Furthermore, one can explicitly discard a first-class constraint and can turn its propagation on or off. We demonstrate these operations in the following application areas:

Early failure detection. Due to the limited view of a single constraint on the constraint store, reasoning and especially failure detection is limited too. Often recognizing a certain constraint pattern makes it possible to spot an inconsistency much earlier than constraint propagation can do and sometimes constraint propagation on its own is not able to detect the inconsistency at all. For example $x < y \ \wedge \ y < x$ is obviously inconsistent. But the time ordinary finite domain propagation takes to detect the inconsistency is proportional to the domain size of x and y, and hence, can be quite long. Reasoning about the constraints themselves can detect the unsatisfiability of this constraint immediately.

Constraint simplification. Constraints fed into a constraint solver can often be improved regarding their propagation behavior. Common sub-constraints, for example, can be collapsed and constraints can be reformulated to provided for better domain pruning.

Garbage Collection. Usually constraints are garbage collected as soon as they are entailed by the constraint store. But typically that requires the parameter of the constraints to be determined. In many cases constraints could be discarded earlier. Consider the finite domain constraint $x + 1 = z \wedge x \leq z$. The constraint $x \leq z$ can be discarded since it is implied by $x + 1 = z$.

Minimal Sets of Inconsistent Constraints. Like every kind of programming, constraint programming is prone to error. A common programming error is to put up an incorrect model a given problem or to implement a constraint model incorrectly. This frequently results in inconsistent constraints which cause immediate failure. Debugging such symptoms is supported by finding sets of constraints that are responsible for the inconsistency.

First-class constraints are defined as an abstract data type, i. e., in terms of operations on them. They are true first-class citizens: they can occur at any position where primitive values can occur too, e. g., as parameters of applications, as return values of functions, or as parts of composite data structures. That makes the new powerful programming techniques possible and allows the programmer, for example, to combine constraint inference on variable values with rewriting techniques to implement hybrid constraint solvers. Furthermore first-class constraints can be used for prototyping sophisticated new constraints.

To our knowledge existing systems do not provide first-class constraints even though it is straightforward to add them to existing solvers (cf. Sect. 8). It is not sufficient to

have access to a C++ object representing a constraint as in ILOG Solver [14, 7]. A first-class constraint is a value of an abstract data type defined by a set of operations (cf. Sect. 2).

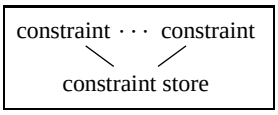
First-class constraints have been implemented with Mozart Oz [9] and the extensions are orthogonal to the existing solver and do not impose any performance penalty when not using first-class constraints.

Plan of the paper. Sect. 2 defines first-class constraints as abstract data types. The following sections investigate early failure detection, simplification, garbage collection of constraints, and finding minimal sets of consistent constraints. Sect. 7 contrasts the expressiveness of first-class constraints with reified constraints, Sect. 8 discusses implementation issues and Sect. 9 comments on related work. The paper closes with concluding remarks.

2 Constraints as First-Class Values

This section introduces a general model for constraint inference serving as a base for the promotion of constraints to first-class status. Then first-class constraints are introduced as values of an abstract data type.

A Model for Constraint Inference. Constraint inference involves a *constraint store*, holding so-called *basic* constraints. A basic constraint is of the form $x = v$ (x is bound to a value v), $x = y$ (x is equated to another variable y), or $x \in D$ (x takes its value in D). Attached to the constraint store are *non-basic* constraints. Together with the constraint store they form a *computation space*. A computation space can be asked, among other things, if propagation has reached a fix-point [16].



Non-basic constraints, as for example "$x + y = z$", are more expressive than basic constraints and, hence, require more computational effort. In the following we call a non-basic constraint "constraint". A constraint is realized by a computational agent (a so-called *propagator*) observing the basic constraints of its *parameters* (which are variables in the constraint store; in the example x, y, and z). The purpose of a constraint is to infer new basic constraints for its parameters and add them to the store. A constraint terminates (fails) if it is inconsistent with the constraint store or if it is explicitly represented by the basic constraints in the store, i. e., it is entailed by the store. A computation space becomes entailed as soon as all constraints are entailed or it becomes failed as soon as at least one constraint fails.

First-Class Constraints. A first-class constraint is a value of an abstract data type and is hence defined in terms of its operations. It can be handled like any other primitive value, i.e, it can be part of composite data structures or can be used in applications or expressions.

Operations on first-class constraints are provided by the module `Constraint`. Access to operations is obtained by the "`.`"-operator and operations are applied by the "`{}`"-operator.

Note that reflective operations are typically *non-monotonic*, i. e., the produced result depends on the current state of the solver. Hence, these operations can be safely applied only if propagation has reached a fix-point. This has to be taken into account when adding new basic constraints to the constraint store while reasoning over first-class constraints. Adding new basic constraints typically requires the recomputation of the fix-point resulting in a changed set of first-class constraints to reason about.

First we define a minimal set of operations, i. e., this set does not contain operations which can be expressed by other operations of this set. Then we introduce operations that make more concise and elegant programming possible.

The following operations designate the minimal set of operations to be provided:

The first two operations are required to obtain access to a first-class constraint and to be able to identify a value as a first-class constraint.

- `C <- { F }` (for short `<-`-operator) creates the constraint `F`, adds it to the current computation space, and binds `C` to an abstract value referring to `F`.
- `C <-# { F }` (for short `<-#`-operator) creates the constraint `F`, adds it inactive, i. e., the propagation is turned off, to the current computation space, and binds `C` to an abstract value referring to `F`. The `<-#`-operator is used in conjunction with the following abstraction:
- `{Constraint.activate C}` turns constraint propagation of constraint `C` on.
- `{Constraint.is C B}` binds `B` to **`true`** if `C` refers to a constraint and otherwise to **`false`**.

Obviously `C <- { F }` can be expressed by combining `C <-# { F }` and `{Constraint.activate C}` but it is added for convenience since it is the usual way to create a first-class propagator.

Programming with first-class constraints typically involves rewriting sets of constraints to operationally more efficient formulations (the most efficient one is of course **`true`**). That requires discarding the redundant constraint which is replaced. Furthermore, reasoning about constraints may take into account that a constraint has already become entailed by the constraint store, i. e., can be ignored.

- `{Constraint.discard C}` discards `C` explicitly, i. e., `C` is removed from the computation space. By discarding a constraint, its whole host space may become entailed.
- `{Constraint.isEntailed C B}` binds `B` to **`true`** if `C` is entailed, either explicitly by the operation `discard` or by entailment through the constraint store, and otherwise to **`false`**.

To be able to reason about constraints the programmer needs to identify what kind of constraint she is dealing with and what the parameters of the constraints are like. The question of which parameters are equal is especially interesting because it makes reformulations of constraints possible.

- `{Constraint.getName C N}` binds `N` to the name of `C`.
- `{Constraint.getParameters C Ps}` binds `Ps` to the parameters of `C`.

– `{Constraint.identifyParameters Vs Ids}` maps the list of variables `Vs` to a list of integer identifiers `Ids` by assigning to each element in `Vs` the index of its first occurrence in `Vs`. Thus equal variables can be detected easily.

Additionally, we propose operations that have turned out to be useful and convenient in the applications discussed in this paper.

– `{Constraint.toString C S}` binds `S` to a textual representation of `C`.
– `{Constraint.reflectSpace Rs Cs}` takes a list `Rs` of variables. It collects all propagators that have at least one variable of `Rs` as a parameter. Furthermore, it collects propagators which share parameters with collected propagators. Thus, the transistive closure of all propagators "reachable" from `Rs` is computed. The collected propagators are turned into some normal form and returned in the list `Cs`.

The application of `Constraint.reflectSpace` makes it possible to use first-class constraints in an orthogonal way since the original constraint program needs not be modified (cf. Sect. 3 and Sect. 4).

3 Early Failure Detection

One of the major goals of constraint programming is to avoid exploration of parts of the search tree that do not contain any solutions. But there are cases where propagation takes significant time to detect failure or is even unable to do so. An example for potential long lasting propagation are the finite domain constraints $x < y \wedge y < x$ and $2x = y \wedge 2u = v \wedge y + 1 = v$ assuming sufficiently large domains.[1] An example for an unsatisfiable constraint that cannot be spotted without any meta reasoning is $x, y, z \in \{0, 1\} \wedge x \neq y \wedge x \neq z \wedge y \neq z$.

This section demonstrates how meta constraint programming can be used to detect unsatisfiable constraints where ordinary constraint propagation fails to do so. Thus the search tree can be significantly pruned and bigger instances of the problem can be solved.

We use as example a modified Hamiltonian path problem, where the aim is to find a path through a given directed graph from an arbitrary starting node to an arbitrary ending node such that all nodes of the graph are visited once and the path is valid for the reverse direction too.

The Constraint Model and its Implementation. The problem data is given as set $Arcs$ of 2-tuples $arc(f, T)$, where the set $T \subseteq \{1, \ldots, n\}$ contains all nodes $t \in T$ such that there is an arc from node f to t. Every of the n nodes of the graph is represented by a finite domain variable $x_i \in \{1, \ldots, n\}$ which represents the position of the ith node; the variables have to be pairwise distinct (constraint (1)). Constraint (2) expresses the path from the starting node to the ending node. Node x_i is the successor of x_f if $x_i = x_f + 1$ holds. Note the extra clause for the ending point. The constraint (3) is dual to constraint (2) and models the reverse path.

[1] Due to the significant propagation time, we used these constraints in [13] to benchmark the propagation performance of our constraint solver.

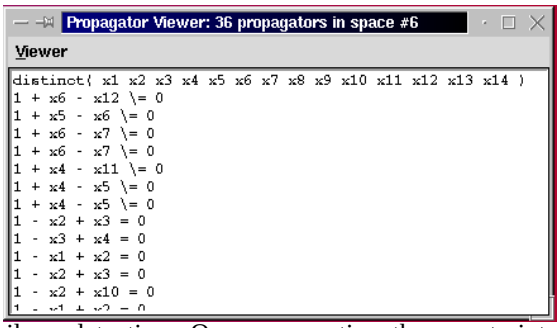
$$distinct(x_1, \ldots, x_n) \tag{1}$$

$$\forall\ arc(f,T) \in Arcs : \bigvee_{i \in T} (x_i = x_f + 1) \vee x_f = n \tag{2}$$

$$\forall\ arc(f,T) \in Arcs : \bigvee_{i \in T} (x_i + 1 = x_f) \vee x_f = 1 \tag{3}$$

We have implemented the constraint model one-to-one with Mozart Oz finite domain constraints and used disjunctive combinators producing choice-points to obtain the same behavior as the program used in [4]. The search strategy is naïve, i. e., it picks from the left-most finite domain variable x_l the minimum element m and creates a choice-point $x_l = m \vee x_l \neq m$.

Deriving an Early-Failure Criterion. Deriving a criterion is a creative process and it is hard to give any guidelines. But it is helpful to have a tool handy that displays the constraints in a node of the search tree. Mozart Oz [9] offers a combination of such tools, namely the Oz Explorer [15] and the Oz Propagator Viewer.[2]



The figure shows a part of the constraints of a node of the search tree without early failure detection. One may notice the constraints $distinct(\ldots, x_3, \ldots, x_{10}, \ldots) \wedge 1 - x_2 + x_3 = 0 \wedge 1 - x_2 + x_{10} = 0$ (last two lines). Substitution of the two equations yields $x_3 = x_{10}$, which contradicts the constraint $distinct(\ldots, x_3, \ldots, x_{10}, \ldots)$ (top line). Generalization of this observation leads to an early failure detection criterion: the set D contains all indices of variables required to be pairwise distinct (derived from the parameters of the $distinct$-constraint). The criterion is: $\exists C_1, C_2 : C_1 \equiv 1 + a_i x_i + a_j x_j = 0 \wedge C_2 \equiv 1 + a_k x_k + a_l x_l = 0 \wedge a_{i,j,k,l} \neq 0 \wedge i = k \wedge j \neq l \wedge j, l \in D \wedge a_j = a_l \rightarrow$ failure.

Adding the Early Failure Detection Criterion. The early failure detection code is completely factored out. It is embedded in the procedure `DetectFailureEarly` which is applied as soon as constraint propagation reaches a fix-point, i. e., right before the creation of a new choice-point.[3] The procedure reflects the constraints to their first-class representation `Cs` according to a normal form. The variable `EqCs` refers to the equational constraints and the variable `DistinctCs` to the pairwise distinct constraints. Then

[2] The Propagator Viewer is still experimental and not yet official part of the Mozart Oz distribution. It can be obtained from the author.

[3] Mozart Oz provides means to synchronize on reaching a propagation fix-point: A unary procedure can be passed to the search engine and this procedure is applied to the solution variable of a search problem as soon as a fix-point is reached.

for each $distinct$-constraint a set D is computed and stored in the list of sets values `DistinctSets` (see [12] for details on integer sets in Mozart Oz). Here the implementation is more general than required for this example.

```
proc {DetectFailureEarly RootVars}
   Cs          = {Constraint.reflectSpace RootVars}
   EqCs        = {FilterEqualityConstraints Cs}
   DistinctCs  = {FilterDistinctConstraints Cs}
   DistinctSets = {ComputeDistinctSets DistinctCs}
```

Then two nested loops (procedures `ForAllTail`[4] and `ForAll`[5] applying anonymous procedures `$`) try to match the appropriate equational constraints according to the early failure detection criterion. An equational constraint is represented by a tuple `´=:´(P LHS RHS)` where `P` is a reference to the actual constraint and `LHS` (`RHS`) is the left hand-side (right hand-side) of the equation. The left resp. right hand-side is represented by a list of addend tuples `addend(Sign Coeff Var)` where `Sign` is the sign (-1 or 1), `Coeff` is the absolute value of the coefficient, and `Var` is a reference to the variable.

Constraints of form $1 + ax + by = 0$ are isolated by pattern matching and the pattern for such a constraint is `´=:´(_ [A1 A2 A3] 0)`[6] as it can be found in the **`case`**-statements.

```
in
   {ForAllTail EqCs
    proc {$ Tail}
       case Tail of  (´=:´(_ [A X1 X2] 0)) | T then
          {ForAll T
           proc {$ TC}
              case TC of ´=:´(_ [B Y1 Y2] 0) then
```

After isolating two matching equational constraints the constant addends are compared and it is checked if the variables are in a D-set. The predicate `Some` is true if at least one of the elements of the list passed (here `DistinctSets`) evaluates the 2nd argument function to **`true`**.

```
                if A == B andthen
                   {Some DistinctSets
                    fun {$ Set}
                       {VarIsInSet X1 Set}
                       andthen {VarIsInSet X2 Set}
                       andthen {VarIsInSet Y1 Set}
                       andthen {VarIsInSet Y2 Set}
                    end}
                then
```

[4] The procedure `{ForAllTail List Proc}` applies the unary procedure `Proc` to all non-`nil` tails of list `List`.

[5] The procedure `{ForAll List Proc}` applies the unary procedure `Proc` to all elements of list `List`.

[6] Note that there is an order on the addends: the first one is constant, the next ones contain variables and the variables are subject to a certain order.

Here the anonymous function `$` checks if the variables of the addends are in one and the same D-set. It uses the predicate `VarIsInSet` which checks if a variable is in a given set. The connector **`andthen`** is a short-circuit conjunction.

```
                    if {IsEqAddend X1 Y1}
                            andthen {IsNeqAddend X2 Y2}
                       orelse {IsEqAddend X1 Y2}
                            andthen {IsNeqAddend X2 Y1}
                       orelse {IsEqAddend X2 Y1}
                            andthen {IsNeqAddend X1 Y2}
                       orelse {IsEqAddend X2 Y2}
                            andthen {IsNeqAddend X1 Y1}
                    then fail % raise failure
                    end
                end
            end % case
        end}
    end % case
end}
end % DetectFailureEarly
```

Finally, the variables of the addends are tested to meet the early failure detection criterion and if so, failure is raised by the statement **`fail`**. The predicate `IsEqAddend` (`IsNeqAddend`) tests if two addends are equal (not equal). The individual applications of `IsEqAddend` are connected by the short-circuit disjunction `orthen`.

Evaluation. Table 1 shows the effectiveness of the presented technique impressively. Entries '-' indicate that after 100.000 nodes of the search tree no solution was found and search was aborted.

# nodes	no early failure detection solution found after # choices/# failures	with early failure detection solution found after # choices/# failures	# detected failures
10	72/52	72/52	0
20	-	160/124	1
30	-	298/244	68
40	-	499/406	162
50	-	499/406	162

Table 1. Effectiveness of early failure detection.

By accident the results for problems with 40 and 50 nodes are identical. The first solution was found on a 200MHz Pentium Pro in a range from a tenth of a second till less than a minute depending on the problem. But the benchmarks aim at demonstrating the effectiveness of the technique, and the early failure detection code has not been particularly optimized.

Early failure detection requires constraints to be first-class values in order to reflect the state of the constraint solver for making symbolic detection of inconsistent constraints possible.

4 Constraint Simplification

This section demonstrates another constraint programming technique made possible by first-class constraints. It is not unusual that a constraint model and consequently its implementation contains redundant constraints or constraints in a formulation that does not allow for the strongest possible propagation.

Consider the constraint $x+x=y \wedge x \in \{1,2\} \wedge y \in \{3,4\}$. Without exploiting the equality of the two variables on the left hand-side the constraint cannot deduce that the only valid instantiation is $x=2 \wedge y=4$. Hence the simplification $x+x=y \rightarrow 2x=y$ improves constraint propagation significantly.

This section reuses the Hamiltonian path problem defined in Sect. 3 but uses reified constraints instead of disjunctive combinators. A reified constraint connects a constraint C with a 0/1-variable B: $(C \leftrightarrow B) \wedge B \in \{0,1\}$. Variable B is bound to $1(0)$ if C is entailed (disentailed). As long as B is unbound C does not add any constraints to the constraint store. In case B is bound to $1(0)$ the reified constraint is replaced by $C(\neg C)$. Reified constraints are used mainly for handling over-constrained problems, i. e., problems where not all constraints can be fulfilled at once, or for modeling disjunctive constraints as in the following case.

The Constraint Model and its Implementation. The constraint model expresses the disjunctions by reified constraints. The parentheses "()" enclosing the equations indicate reification. Constraint (5) stands for the path from the starting to the ending node and constraint (6) for the same path in reverse direction.

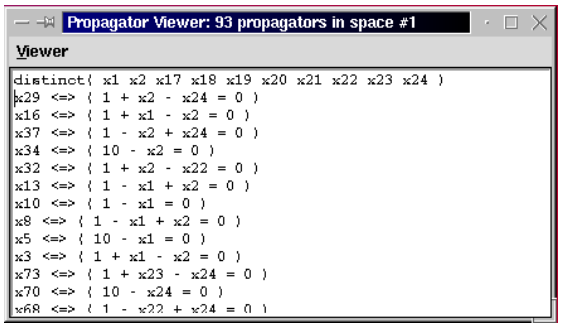
$$distinct(x_1, \ldots, x_n) \tag{4}$$

$$\forall\ arc(f,T) \in Arcs : \left((x_f = n) + \sum_{i \in T} (x_i = x_f + 1) \right) = 1 \tag{5}$$

$$\forall\ arc(f,T) \in Arcs : \left((x_f = 1) + \sum_{i \in T} (x_i + 1 = x_f) \right) = 1 \tag{6}$$

Deriving a Simplification Rule. Regard the lines in the figure starting with the variables x_{16} and x_3. In both cases the corresponding constraints reify $1 + x_1 - x_2 = 0$. That makes it possible to equate x_{16} and x_3 and to discard a copy of $1 + x_1 - x_2 = 0$. In general $\exists (C_i \leftrightarrow B_i), (C_j \leftrightarrow B_j) : C_i = C_j \rightarrow B_i = B_j \wedge discard(C_j)$.

In this case finding a suitable rule is easy.



The proposed simplification
has two effects: it removes redundant propagation by discarding superfluous constraints, and it strengthens the constraint store by adding equality constraints.[7]

Adding Constraint Simplification. Constraint simplification is executed whenever propagation reaches its fix-point. It reflects the constraints of a computation space with `Constraint.reflectSpace` to obtain direct access to the constraints, and function `FilterReified` filters out all reified constraints $(C \leftrightarrow B)$ since the other constraints are of no interest. The result is stored in `ReCs`. Furthermore, `FilterReified` generates a textual representation of C using `Constraint.toString` which is used as index for the dictionary `Dict` to easily identify reified constraints which are identical modulo the 0/1-variable B.

```
fun {SimplifyAndCollect RootVars}
   ReCs = {FilterReified {Constraint.reflectSpace RootVars}}
   Dict = {NewDictionary}
in
```

For each reified constraint the actual simplification is done in a `ForAll` loop which calls an anonymous procedure `$`. This procedure accesses the components of its argument by pattern matching: `I` is the textual representation index, `P` is a reference to the reified constraint itself, `C` the reified constraint, and `B` is a 0/1-variable. Note that `#` is the infix tuple constructor and hence `I#reified(P C B)` is a 2-tuple matched against the argument passed to the anonymous procedure.

```
   {ForAll ReCs
    proc {$ I#reified(P C B)}
       if {Dictionary.member Dict I} then
          reified(P1 C1 B1) = {Dictionary.get Dict I}
       in
          B1 = B
          {Constraint.discard P}
       else
          {Dictionary.put Dict I reified(P C B)}
       end
    end}
   % return 0/1-variables of the reified constraints
   {RetrieveBools Dict}
end % SimplifyAndCollect
```

Using `Dictionary.member` the procedure checks if a reified constraint is already stored under the textual representation index `I`. If so, the individual components of the entries are retrieved by pattern matching[8], the 0/1-variables are equated, and the constraint referred to by `P` is stated to be entailed by `Constraint.discard`. That is exactly what the simplification rule requires. In case the reified constraint is not yet

[7] In Mozart Oz equality is represented directly in the constraint store.

[8] The return value of the function application `{Dictionary.get Dict I}` is matched against the tuple `reified(P1 C1 B1)` and the newly introduced variables `P1` `C1` `B1` are bound accordingly.

stored in `Dict` a new entry is created by `Dictionary.put`. Finally, the 0/1-variables of the reified constraints are retrieved and returned by `RetrieveBools`.

The search strategy branches over the 0/1-variables of the reified constraints (5) and (6) returned by `SimplifyAndCollect` to stay as close as possible to the program used in Sect. 3.

Evaluation. The number of 0/1-variables coming from the reified constraints is significantly reduced by simplification. In combination with the additional equality constraints, this leads to an enormous reduction of choice points (see Table 2), even better than for early failure detection in Sect. 3.

# nodes	no simplification solution found after # choices/# failures	with simplification solution found after # choices/# failures	# simplified constraints
10	292/288	4/2	26
20	-	19/0	60
30	-	19/94	118
40	-	2673/2632	158
50	-	122/73	199

Table 2. Effectiveness of constraint simplification.

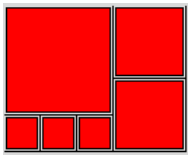
Only for the graph with 40 nodes the number of choice points is much greater. This indicates that the search strategy used is not stable enough against variations of the problems, but this is not the focus of this paper.

Constraint simplification requires constraints to be first-class values in order to reflect the state of the constraint solver and thus making symbolic constraint simplifications possible.

5 Garbage Collection of Constraints

Usually constraint solvers collect redundant constraints as they become entailed by the constraints in the store. Even if their memory is not freed due the implementation of the solver, they are at least not rerun anymore if their parameters receive new basic constraints. For example, consider $x \leq y \wedge x \in \{0,\ldots,3\} \wedge y \in \{3,\ldots,6\}$, and suppose the $\leq$-constraint is entailed by the basic constraints $x \in \{0,\ldots,3\} \wedge y \in \{3,\ldots,6\}$ and is garbage collected. That is not always the case, as demonstrated for the no-overlap constraint in the tiling problem.

The Problem Description and the Constraint Model. A given number of square tiles has to be placed on a master plate (see figure). The tiles must not exceed the master plate along the x- and y-axis. This is enforced by the $capacity$-constraint which is not of interest here. Furthermore, the tiles must not overlap which is enforced by the $nonoverlap$-constraint. Consider the square tiles T_1 and T_2 with length l_1 and



l_2. Their positions on the master plate are determined by their left lower corners (x_1, y_1) and (x_2, y_2) which results in the $nonoverlap$-constraint

$$x_1 + l_1 \leq x_2 \vee x_2 + l_2 \leq x_1 \vee y_1 + l_1 \leq y_2 \vee y_2 + l_2 \leq y_1. \tag{7}$$

The constraint (7) is encoded by the reified constraint

$$(x_1 + l_1 \leq x_2) + (x_2 + l_2 \leq x_1) + (y_1 + l_1 \leq y_2) + (y_2 + l_2 \leq y_1) \geq 1.$$

Note the $\geq$-constraint which is necessary since two tiles can be non-overlapping in both the x- and y-axis. This constraint causes the trouble regarding garbage collection since as soon as one of its reified constraints is valid the remaining three constraints could be discarded. But this is impossible without first-class constraints because a reified constraint cannot be discarded, it just reduces to its embedded positive or negative constraint.

Implementation Issues. The encoding of the $nonoverlap$-constraint by the procedure `ImposeNonOverlap` catches the references to the individual constraints involved, i. e., the reified $\leq$-constraints and the $\geq$-constraint. This is achieved by using the `<--`operator. The $nonoverlap$-constraint returns these references wrapped in the tuple `nonoverlap(P0 [P1 P2 P3 P4])`.

```
fun {ImposeNonOverlap X1 Y1 L1 X2 Y2 L2}
   B1 B2 B3 B4 P0 P1 P2 P3 P4
in
   P1 <- {(X1 + L1 =<: X2) = B1}
   P2 <- {(X2 + L2 =<: X1) = B2}
   P3 <- {(Y1 + L1 =<: Y2) = B3}
   P4 <- {(Y2 + L2 =<: Y1) = B4}
   P0 <- {B1 + B2 + B3 + B4 >=: 1}

   nonoverlap(P0 [P1 P2 P3 P4])
end
```

The procedure `CollectNonOverlapConstraints` is called when propagation has reached a fix-point. It receives as its argument a list of tuples produced by `ImposeNonOverlap` and checks for all tuples if the enclosed $\geq$-constraint is entailed by applying `Constraint.isEntailed` to `P0`. If so, the remaining reified constraints are determined to be entailed by `Constraint.discard`.

```
proc {CollectNonOverlapConstraints NonOverlapConstraints}
   {ForAll NonOverlapConstraints
    proc {$ nonoverlap(P0 Ps)}
       if {Constraint.isEntailed P0} then
          {ForAll Ps proc {$ P}
                        if not {Constraint.isEntailed P}
                        then  {Constraint.discard P} end
                     end}
```

```
            end
         end}
   end
```

Evaluation. Table 3 shows the number of reified $\leq$-constraints garbage collected for different instances of the tiling problem.

# tiles	# collected constraints	saved memory
6	18	5K
9	44	11K
17	197	100K
21	1528	1.7M

Table 3. Effectiveness of constraint garbage collection.

The third column shows the amount of memory saved which is in balance with the overhead imposed by the extra data structures used.

The proposed garbage collection scheme relies on first-class constraints for detecting redundant constraints and for explicitly discarding such constraints since they cannot be garbage collected yet by entailment.

6 Computing Minimal Sets of Inconsistent Constraints

Solving a combinatorial problem by constraint programming requires expressing the problem in terms of constraints, i. e., finding a constraint model, and implement the conceived constraint model as a constraint program for a concrete constraint solver. This process is prone to error. The first run of the constraint program frequently results in an immediate failure of the solver, caused by an inconsistent set of constraints. The set of constraints is usually large and only a subset is responsible for the failure. Hence, being able to find minimal inconsistent subsets of constraints of a given set of constraints may simplify debugging the above-described situation significantly. Note that there may be several minimal inconsistent sets of constraints since several errors may occur at once.

Idea. According to the model presented in Sect. 2, constraint propagation takes place in computation spaces. A failed space hosts an inconsistent set of constraints. Finding minimal sets of inconsistent constraints starts by reflecting the last consistent state of the failed computation space. Reflection comprises all basic constraints, i. e., constraints of the form $x \in D$ and $x = y$, and all propagators. Such a reflection makes it possible to restore the constraints of the last consistent state of the failed space in a fresh space. Since such a restoring would immediately result in a failure, the propagators are imposed as *inactive first-class propagators*, i. e., they are imposed with propagation turned off. A solution is a set of inconsistent propagators. Hence, the fresh space is appropriately wrapped to propagate its failure as solution. The starting point for searching a solution that all propagators are inactive. Search turns propagation successively on for every propagator and then checks whether the set of active propagators is inconsistent or not.

Implementation. The last consistent state of a failed computation space is reflected by `Constraint.reflectSpace` and equal variables are spotted by `Constraint.identifyParameters`. The reflected propagators are imposed as inactive first-class propagators using the `<-#`-operator in a fresh computation space to be able to catch failure. Furthermore, every propagator is assigned a unique integer identifier and is connected via its corresponding first-class value to a $0/1$-variable such, that constraining the $0/1$-variable to 1 (0) activates (discards) the propagator. That makes it possible to use the standard search library on finite domain variables. A possible solution is a set of integers denoting a set of inconsistent propagators.

We are interested in minimal sets of inconsistent constraints. Branch-and-bound search is used to ensure that new solutions are either a proper subset of an already found solution or a distinct set of inconsistent constraints. An appropriate order constraint, which ensures that new solutions meet the above condition, has to take into account two cases:

1. a new solution N is a proper subset of the current solution O, i. e., $N \subset O$ resp. $N \setminus C \neq \emptyset$ where $C = N \cap O$.
2. a new solution N is distinct to the set of inconsistent constraints O, i. e., $O \setminus C \neq \emptyset \wedge N \setminus C \neq \emptyset$ where $C = N \cap O$.

These two conditions can be collapsed to $O \setminus (N \cap O) \neq \emptyset$. The implementation of the order constraint uses Mozart Oz's finite set constraints [12].

A first minimal set of inconsistent constraints is found by starting with all propagator's propagation turned on and successively turning propagation off. As soon as turning a propagator inactive makes the set of active propagators not immediately inconsistent, this propagator is kept active. Thus, by processing all propagators once, a first minimal set of inconsistent constraints is found. Finding other possible sets requires backtracking to the first propagator turned inactive and turning this propagator active and continue search from there. The order constraint described above prunes the search space further by disallowing solutions subsuming already known ones.

Example. Consider the inconsistent set of constraints composed of the constraints $c_{1\ldots5}$: $x <_{c1} y \wedge y <_{c2} z \wedge z <_{c3} x \wedge z <_{c4} u \wedge u <_{c5} x$. As one can easily see, there are two minimal sets of constraints that are not in a subset relation: $S1 = \{c1, c2, c3\}$ and $S2 = \{c1, c2, c4, c5\}$.

Expectedly, the search routine finds two solutions $S1$ and $S2$, as the corresponding search tree shows (see the rhombus-shaped nodes in the Mozart Explorer display [15] in Fig. 1).

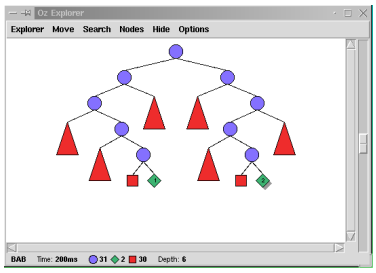
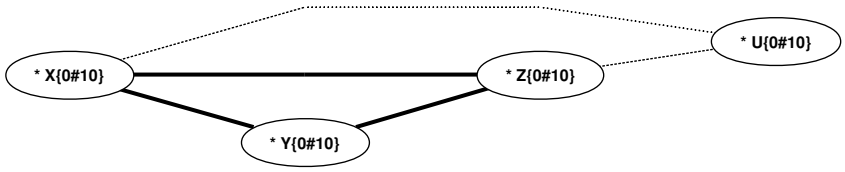
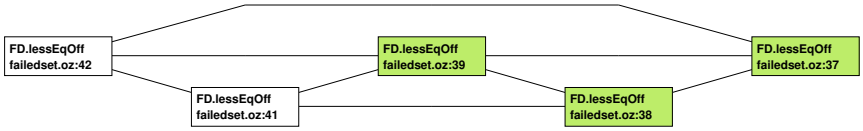


Fig. 1. Search tree of example.

We use the Mozart Constraint Investigator [10] to present the solutions graphically. The first solution, corresponding to $S1$, is shown as a variable graph in Fig. 2(a), i. e., the nodes of the graph denote variables and edges represent propagators. Thick solid edges stand for propagators being part of the set of inconsistent constraints.



(a) Parameter graph where failed propagator edges are thick solid line.



(b) Constraint graph where nodes of failed propagators are shaded.

Fig. 2. First solution (left-most solution (rhombus) node in Fig. 1).

The propagator graph in Fig. 2(b) depicts propagators as nodes and variables shared between propagators as edges between the respective propagators. Propagators being part of the inconsistent set of constraints are shaded. Having propagators, represented by their nodes, identified as being responsible for a failure, the Investigator allows to highlight their occurrence in the source code by simply clicking the respective propagator node.

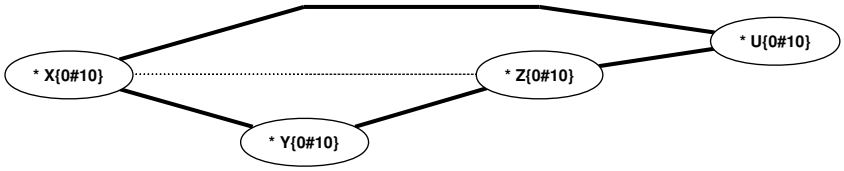


Fig. 3. Variable graph of second solution (right-most solution (rhombus) node in Fig. 1).

The second solution, corresponding to $S2$, is shown as the variable graph in Fig. 3 and reveals a second reason for the example set constraints being inconsistent.

First-class constraints are the key to this application since they make it possible to reflect a failed computation space and to do search over constraints by explicitly turning propagation on and off.

7 First-Class Constraints vs. Reified Constraints

This section summarizes the unique features of first-class constraints used in the presented applications and argues why the expressiveness of first-class constraints goes far beyond what can be expressed with reified constraints.

Reflection. First-class constraints make it possible to reflect the propagator's name and parameters to values. Reified constraints do not offer reflection.

Activation and Deactivation. Applications searching over sets of constraints (cf. Sect. 6) have to be able to impose propagators with propagation turned off and then to toggle propagation as computation proceeds. Initially, reified constraints $(C \leftrightarrow B \in \{0,1\})$ are inactive w. r. t. propagation. Turning propagation on is done by constraining B to 1. But once turned on, propagation cannot be turned off due to monotonicity of reified constraints. [9] Although not mentioned so far, propagation of a first-class constraint `C` can be turned off by calling `{Constraint.deactivate C}`.

Explicit Entailment. Applications rewriting constraints need to be able to discard constraints even if they are not entailed by the constraint store. Replacing a set of constraints $\mathcal{C}$ by $\top$ is the special case of garbage collecting $\mathcal{C}$, as demonstrated in Sect. 5. Replacing $\mathcal{C}$ by $\mathcal{C}' \neq \top$ was discussed as constraint simplification in Sect. 4. Reified constraints do not support this functionality.

Checking for Entailment. It is frequently necessary to find out if a constraint is already entailed or not. First-class constraints provide the operation `Constraint.isActive`. Reified constraints can be used for entailment checking too, where $B = 1$ indicates entailment. The constraint $C \wedge (C \leftrightarrow B \in \{0,1\})$ does this test but it must be ensured that $B \neq 0$.

Operations on first-class constraints are non-monotonic, i. e., they can be undone or can produce different result depending on the current state of the computation space they are applied on. Reified constraints are monotonic, i. e., they *cannot* be undone and in conjunction with a certain set of other constraints they always reach the same fix-point. In fact, first-class constraints and reified constraints are *orthogonal concepts* and reified constraints can of course be first-class constraints too (cf. Sect. 5 where the $nonoverlap$-constraint uses reified first-class constraints). The use of the notion *meta* is somewhat misleading since true meta programming is only possible with the expressiveness that first-class constraints provide.

8 Adding First-Class Constraints to an Existing Solver

This section briefly summarizes the necessary additions to an existing constraint solver to provide for first-class constraints.

Promoting a constraint to first-class status means giving the programmer direct access to it and thus being able to inspect and control the constraint. This is straightforwardly done by introducing a data type referring to constraints.

[9] Note that constraining B to 0 imposes the negative constraint $\neg C$.

Inspecting a constraint (cf. `getName` and `getParameters` in Sect. 2) requires being able to retrieve a constraint's parameters and name. Constraint solvers implemented in C++ typically represent a constraint as an object such that it is easy to add appropriate member functions and keep the changes local to the actual constraints.

Furthermore, first-class constraints need to have a unique identity to enable the check for equality of first-class constraints.[10] This can be done by deriving an identity from the memory address of the object representing a constraint. But care must be taken for garbage collection and all kinds of operations that change the location of a constraint in memory.

Discarding a constraint and checking for entailment (cf. `discard` and `isEntailed` in Sect. 2) typically requires setting a flag in the constraint representation. The constraint solver has to check right before the execution of a constraint whether it was explicitly entailed in the mean-time, i. e., between wake-up and execution, or not.

The programming techniques presented in Sect. 3, Sect. 4, and Sect. 5 need to detect the fix-point of constraint propagation. Hence the constraint language has to provide a combinator that allows the programmer to do so. Implementation may simply check if the propagation queue, which maintains constraints to be executed next, is empty.

The experimental implementation of first-class constraints was straightforward since Mozart Oz provides so-called extensions. They are intended to allow the programmer to add new data types via a C++ interface [8]. There were no modifications necessary to the actual propagation engine such that there are no performance penalties.

9 Related Work

One approach at gaining more control and expressivity over constraints was the idea to exploit a constraint's truth value as proposed for the cardinality constraint in [18]. Applying arithmetic and boolean operations to constraint's truth values was explored in [1]. These constraints are usually called meta or reified constraints. They are available in nearly all current constraints solvers.

Meta-programming as known from Lisp or Prolog means manipulating a program by another program. Therefore, the program code is represented as a term of the respective programming language and then submitted to a meta-interpreter written in this language. Such a scheme for the constraint programming language CLP($\mathcal{R}$) is proposed in [5]. They use `quote` and `eval` functions which are analogous to the corresponding Lisp functions.

Solvers dedicated to a certain set of constraints as well as dedicated constraints can of course do the same analysis as discussed in this paper. In [4] early failure detection as described in Sect. 3 has been proposed as a by-product of analyzing the impact of simplifications for equational constraints on the propagation behavior.

ILOG Solver [7] is a C++ library for constraint programming in C++. It does not support first-class constraints as presented in this paper but ILOG Solver 4.4 allows the user to define a new constraint by defining a new class of constraints derived from

[10] This paper ignores identity on first-class constraints. But if one has to implement `reflectSpace` just with the other operations, to guarantee termination one has to check equality of constraints.

the library class `IlcConstraintI`. It is straightforward to provide the required extra functionality according to Sect. 8 by adding appropriate member functions to the class definition of the new constraint.

Constraint Handling Rules (CHR) [3] are a committed-choice language for rewriting constraints towards a solved form which eventually denotes a solution. A CHR program is a set of guarded rules of the form $H \;\; op \;\; G \mid B$ where $op \in \{$**<=>**, **==>**$\}$, $H = H_1, \ldots, H_i$, $G = G_1, \ldots, G_j$, and $B = B_1, \ldots, B_k$. A multi-head H is a sequence of CHR, the guard G is a sequence of built-in constraints, and the body B is a sequence of CHR and built-in constraints. A rule fires as soon as a the CHR store implies H and the constraint store implies G. Then the CHR and constraint store are extended by B. A propagation rule ($op =$ **==>**) extends the appropriate stores by redundant constraints B. A simplification rule ($op =$ **<=>**) behaves like a propagation rule but additionally removes H from the CHR store. CHR can be used to implement the techniques proposed in Sect. 3 and Sect. 4 due to the multi-heads of the rules. For example, the inconsistent constraint $x < y \wedge y < x$ can be detected by the following CHR rule:

```
less(x,y),less(y,x) <=> true | false.
```

To the best of our knowledge none of the above-mentioned approaches, nor other existing systems, offer the same expressiveness or generality as the scheme proposed in this paper, to promote constraints to first-class status.

10 Conclusion and Future Work

We have introduced constraints as first-class citizens and investigated possible fields of application. Furthermore, we have demonstrated programming techniques using first-class constraints, have proved their effectiveness, and argued that first-class constraints and reified constraints are orthogonal concepts (cf. Sect. 7).

The experiments have shown that the programmer needs appropriate analysis tools to find powerful meta-constraint propagation rules especially for the techniques discussed in Sect. 3 and Sect. 4. Furthermore, the effects of simplification and garbage collection may overlap since simplified constraints become redundant and can be discarded.

The experiments were done with Mozart Oz using the Oz Explorer and the Oz Propagator Viewer. The experimental implementation of first-class constraint was straightforward since Mozart Oz provides adequate programming interfaces to extend the constraint solver's functionality easily from user level [8, 11].

Extending an existing constraint solver can be done with minimal effort and without performance penalties when first-class constraints are not used.

Acknowledgements. I am grateful to Warwick Harwey for discussing with me issues of early failure detection and pointing me to the Hamiltonian path problem as a suitable example. Katrin Erk, Leif Kornstaedt, Kevin Ng Ka Boon and Christian Schulte gave helpful comments on earlier versions of the paper. Furthermore, Christian brought the tiling example in Sect. 5 to my attention. Moreover, I am grateful to Ulrich Neumerkel

for discussing ideas about inconsistent sets of constraints. The graphs in Sect. 6 were drawn with *daVinci* [17]. Last but not least I would like to thank the anonymous referees for their comments.

References

1. Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 1997.
2. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan, December 1988. Institute for New Generation Computer Technology (ICOT),Tokyo, Japan.
3. Thom Früwirth. Theory and practice of constraint handling rules. *Special Issue on Constraint Logic Programming, Journal of Logic Programming*, 37(1–3), October 1998.
4. Warwick Harvey and Peter J. Stuckey. Constraint representation for propagation. In M. Maher and J.-F. Puget, editors, *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming (CP98)*, Lecture Notes in Computer Science, pages 235–249, Pisa, Italy, October 1998. Springer-Verlag.
5. Nevin Heintze, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. Meta-programming in CLP($\mathcal{R}$). *Journal of Logic Programming*, 33(3):221–259, December 1997.
6. Martin Henz and Jörg Würtz. Using Oz for college timetabling. In E.K. Burke and P. Ross, editors, *Practice and Theory of Automated Timetabling, First International Conference, Selected Papers, Edinburgh 1995*, volume 1153 of *Lecture Notes in Computer Science, Springer*, pages 162–178. Springer-Verlag, Berlin-Heidelberg, 1996.
7. ILOG S. A., URL: http://www.ilog.com/. *ILOG Solver 4.4, User's Manual*, 1999.
8. Michael Mehl, Tobias Müller, Christian Schulte, and Ralf Scheidhauer. Interfacing to C and C++. Technical report, Mozart Consortium, 1999. Available at http://www.mozart-oz.org/documentation/foreign/index.html.
9. The Mozart Consortium. *The Mozart Programming System*. http://www.mozart-oz.org/.
10. Tobias Müller. Practical investigation of constraints with graph views. In Konstantinos Sagonas and Paul Tarau, editors, *Proceedings of the International Workshop on Implementation of Declarative Languages (IDL'99)*, September 1999.
11. Tobias Müller. The Mozart Constraint Extensions Tutorial. Technical report, Mozart Consortium, 1999. Available at http://www.mozart-oz.org/documentation/cpitut/index.html.
12. Tobias Müller and Martin Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München, 17–19 September 1997.
13. Tobias Müller and Jörg Würtz. Extending a concurrent constraint language by propagators. In Jan Małuszyński, editor, *Proceedings of the International Logic Programming Symposium*, pages 149–163. The MIT Press, Cambridge, 1997.
14. Jean-François Puget and Michel Leconte. Beyond the glass box: Constraints as objects. In John Lloyd, editor, *Logic Programming – Proceedings of the 1995 International Symposium*, pages 513–527. The MIT Press, Cambridge, December 1995.
15. Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, 8-11 July 1997. The MIT Press, Cambridge.
16. Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, Schloss Hagenberg, Linz, Austria, October 1997. Springer-Verlag, Berlin-Heidelberg.

17. Universität Bremen, Group of Prof. Dr. Bernd Krieg-Brückner. *The Graph Visualization System daVinci*. http://www.informatik.uni-bremen.de/ davinci/.
18. Pascal Van Hentenryck and Yves Deville. The Cardinality Operator: A new Logical Connective for Constraint Logic Programming. In Koichi Furukawa, editor, *Proceedings of the International Conference on Logic Programming*, pages 745–759, Paris, France, 1991. The MIT Press.
19. Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation and evaluation of the constraint language cc(FD). In Andreas Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.