

# Expressive Models for Monadic Constraint Programming

Pieter Wuille and Tom Schrijvers\*

Department of Computer Science, K.U.Leuven, Belgium  
*FirstName.LastName@cs.kuleuven.be*

**Abstract.** This paper presents a new FD-specific modeling front-end for the Monadic Constraint Programming framework for Haskell. A more declarative interface was introduced, supporting reified constraints among others, as well as an optimizing compilation scheme to prevent the inefficiencies that high-level modeling typically introduces. Problems can be solved directly at run-time, or translated to C++ code for later solving. Benchmarks show that solving efficiency approximates that of hand-coded Gecode programs.

## 1 Introduction

The Monadic Constraint Programming framework integrates constraint programming in the functional programming language Haskell [4] as a deeply embedded domain specific language (EDSL).

While the integration is not tied into the language, Haskell does offer good EDSL support to make the embedding quite convenient. Moreover, being less tight does provide for greater flexibility. In addition, the deep embedding of the EDSL allows us to use the constraint model for more than outright solving. For instance, transformations can be applied to the model for generic optimization purposes or to better target a particular constraint solver. Moreover, the model does not have to be solved directly, but can drive a code generator that produces a stand-alone executable for solving at a later time.

This paper reports on the FD-MCP module of the framework, specific to finite domain (FD) solvers. We introduce a more declarative modeling front-end, as well as an optimizing compilation scheme that eliminates the inefficiencies of high-level models.

In contrast to MCP’s generic interface, which is parametric in the constraint domain, FD-MCP[9] provides a finite-domain (FD) intermediate layer. It extends MCP with a common FD-specific syntax, without forcing a particular FD solver or search strategy. Internally the FD layer converts high-level, complex and solver-independent constraints to the low-level constraints supported by a particular FD solver.

On the one hand, this allows the development of solver-independent models, model transformations (e.g., for optimization) and model abstractions (capturing

---

\* Post-Doctoral Researcher of the Research Foundation– Flanders (FWO-Vlaanderen).

frequently used patterns). On the other hand, specific solvers may focus on the efficient processing of their constraint primitives without worrying about modeling infrastructure.

## 1.1 Contributions

This paper presents improvements in the front-end and processing part of FD-MCP.

- **Expression-based modeling language** The concept of constraints has been absorbed into that of expressions. This means that constraints can be used wherever boolean expressions are required and vice versa. This provides for natural support of reified constraints without special syntax.

Furthermore, integer arrays and array expressions have been added. Finally, higher-order combinators (such as `map`, `fold`, `forall`, ...) can be used first-class in these expressions.

The result is a rich, concise and declarative constraint modeling language.

- **Aggressive compilation**

To compile these new high-level expressions to solver-specific constraints, a new compilation scheme was designed and implemented.

Compilation proceeds in separate stages, using an intermediate graph representation, and avoids as much as possible the use of reified constraints or redundant variables. Optimizations are possible at each stage, resulting in far more efficient solving comparable to hand-written low-level models, as benchmarks show.

The generic compilation scheme is presented in Section 2, while the integration with the FD-MCP framework is given in Section 3. Benchmarks were performed to experimentally verify the efficiency, as can be seen in Section 4.

## 1.2 Example Models

As an introductory example, here is the model for the standard *n-queens* problem in FD-MCP:

```

1 model n = exists $ \p -> do           -- request an array p
2   size p @= n                          -- whose size is n
3   p 'allin' (0,n-1)                    -- all of p are in [0..n-1]
4   allDiff p                             -- all of p are different
5   loopall (0,n-2) $ \i -> do          -- foreach i in [0..n-2]
6     loopall (i+1,n-1) $ \j -> do    --   foreach j in [i+1..n-1]
7       (p!i) + i @/= (p!j) + j        --     p[i]+i != p[j]+j
8       (p!i) - i @/= (p!j) - j        --     p[i]-i != p[j]-j

```

To illustrate the use of reified constraints, this is how one can define a count (or cardinality) constraint:

```

1 count col val =
2   csum $ cmap (\v -> channel (v @= val)) col
3
4   count col val @= n ...

```

Here `count` takes an integer array and an integer as input, and returns how often that value occurs in the array. `csum` takes an array of expressions (`cmap (\v -> channel (v @= val)) col` in this case), and returns an expression representing its sum. `cmap` takes an array (`col` here), and a function and returns the array that represents the application of that function on each element of the array. The function `\v -> channel (v @= val)` is a lambda expression that takes an argument `v` and returns the integer 1 when `v` equals `val` and 0 otherwise.

## 2 Compilation scheme

The compilation scheme is responsible for converting (conjunctions of) high-level constraints — represented by boolean expressions over variables — to low-level, solver-specific variables and constraints.

Often there are many ways to translate a high-level model to a low-level one, with varying efficiency for solving. In fact, the genericity of high-level modeling hides many critical aspects from the user, resulting in poor performance when using a naive approach. Optimizing transformations on the model are performed to overcome this.

One reason for poor naive performance is that primitives of the high-level modeling language do not necessarily coincide with those of the low-level constraint system we are translating for. Sometimes, combinations of constraints in the high-level model — connected using auxiliary variables — can be converted to a single more specific constraint for the low-level system. For example, when a global *sum* constraint is provided by the underlying solver, it is typically better to translate  $a = b + e \wedge e = c + d$  to a single  $a = \text{sum}([b, c, d])$  constraint, eliminating the  $e$  variable, unless it is used in other constraints, needed for branching, or used as a solution.

On the other hand, sometimes it may be necessary to introduce additional variables when the low-level system lacks primitives provided by the high-level language.

To tackle this problem in a general way, the model is broken down to an intermediary graph-based format in which all expressions and subexpressions of the original become separate variables, independent from how they were introduced. In a second stage, the pieces are recombined (possibly in a different way) to constraints for the low-level system, by pattern matching on parts of this intermediary graph.

Section 2.1 describes the properties of the expected input model, while the conversion to the broken-down intermediary format is described in Section 2.2. Finally, the conversion to solver-specific constraints is shown in Section 2.3.

## 2.1 The expression tree

The algorithm expects a model as input, represented by an expression tree over variables. It does not matter how this tree is constructed, or exactly which types of nodes it can contain. We assume that basic arithmetic, relational and boolean operators are present, as well as some array operations.

Working in the context of the functional language Haskell, we want higher-order constructs to be first-class. Typically, constraint modeling languages support constructs like loops and sums. Instead, we encode them as higher-order expression nodes in the tree:

- `forall`( $[e_1, \dots, e_n], f$ ) holds if  $f e_1 \wedge \dots \wedge f e_n$  holds. When  $[e_1, \dots, e_n]$  is a sequence of successive values  $[i, i + 1, \dots, n]$ , this corresponds to a `for` loop.
- `fold`( $[e_1, \dots, e_n], z, f$ ) represents the expression  $f(e_1, f(e_2, \dots f(e_n, z) \dots))$ . For instance, `fold`( $[a, b, c], 0, +$ ) is equal to  $a + b + c$ . This construct is also known under the name *reduce*, and can be used to represent a count, a sum, or any other aggregate over an array of variables.
- `map`( $[e_1, \dots, e_n], f$ ) represents the array  $[f e_1, f e_2, \dots, f e_n]$

Semantically, these expressions are equivalent to repeated application of a function to an array of expressions. If we are compiling parametrized models to C++ (see [10]), the size of certain arrays may not be known at Haskell-runtime. By allowing these higher-order expressions to refer to such “delayed” arrays, we do gain expressivity.

These primitives in the expression tree can be constructed by helper functions. For example, `cmap` and `csum` (as used in Section 1.2), produce a `map`(*array*, *f*) and a `fold`(*array*, 0, +) respectively.

## 2.2 Constraint Network Graph

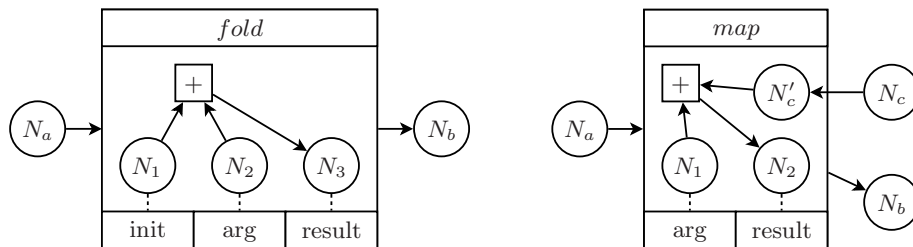
As explained, the input model is transformed to an intermediary format. All subexpressions are replaced by additional variables and constraints. For example,  $a < b + c$  would be broken up into the conjunction  $a < t_1 \wedge t_1 = b + c$ . Since these constraints are recombined to more complex constraints in the next stage, convenient access from a constraint to its variables and vice versa is needed. The original expression tree does not make this information explicit. Instead, the set of broken-down constraints is represented by a graph which does, a constraint network graph. The nodes in this graph are variables, expressions and subexpressions of the constraint model, while the edges are the constraints between them. Since constraints are not necessarily binary, in general the edges are hyperedges and the graph a hypergraph.

To encode higher-order expressions, we allow certain (hyper)edges to be labeled with a sub-model. This has several advantages over replacing them by the sequence of direct constraints they represent (also called flattening or more specifically loop unrolling):

- It allows the additional graph-based optimizations to be performed on the problem class level, before instantiation of parameters.

- Creating a graph of the flattened model may well be very expensive, since its size is proportional to the number of variables and constraints in the instance.
- Unrolling is not possible if loop or array sizes depend on a parameter that is not known at Haskell-runtime.

These internal graphs can have some nodes marked as imported from the parent model. For example, the model  $b \text{ @=} \text{csum } a$  evaluates to the expression  $b = \text{fold}(a, 0, +)$ , and the model  $b \text{ @=} \text{cmap } (\lambda x \rightarrow x + c) a$  evaluates to  $b = \text{map}(a, +c)$ . The corresponding constraint network graphs are shown in Figure 1. The boxes and arrows represent the hyperedges/constraints, while the circles represent the nodes/variables. The boxes for *fold* and *map* are labeled with a new graph, with some nodes marked specially (*init, arg* and *result*). Since the variable  $c$  is not local to the *cmap*'s function argument (like  $x$ ), it is imported from the parent model's  $N_c$  as  $N'_c$ .



**Fig. 1.** submodels:  $b = \text{fold}(a, 0, +)$  and  $b = \text{map}(a, +c)$

Translation from an expression tree to such a graph goes as follows:

- For every node in the tree a corresponding node in the graph is created. To preserve variable identities, variables in the tree are reused as their corresponding nodes in the graph.
- The relation between a tree node and its children is materialized as an edge between the corresponding graph nodes.

For example, consider the expression  $a + b$ . The corresponding graph contains three nodes:  $N_{a+b}$ ,  $N_a$  and  $N_b$ , with a hyperedge labeled  $+$  among them.

Note that multiple occurrences of the same variable in the tree end up sharing the same node in the graph, abandoning the tree invariant. The possibility of sharing is further exploited by mapping identical subtrees onto the same graph node. This essentially results in CSE (common subexpression elimination), which is more thoroughly explored in [5].

Note that in the model, constraints are represented by boolean expressions. Their return values denote their truth value. This means that they are all explicitly reified in the constraint network form. Since the point of a CSP (Constraint

Satisfaction Problem) is finding an assignment for the variables which causes the model expression to evaluate to true, the truth value for the root node incurs an additional constraint, equating it to the constant “true”.

Clearly, we do not want everything to be translated to reified constraints. These are often less efficient, or may not even be supported by solvers.

A first step in preventing this is *constant propagation*. When a model consists of  $c_1(a, b) \wedge c_2(a, b)$ , a naive translation creates a constraint network with three nodes for boolean variables in addition to  $a$  and  $b$ :  $v_{12}$ ,  $v_1$  and  $v_2$ . The generated constraints are  $v_{12} = true$ ,  $v_{12} = v_1 \wedge v_2$ ,  $c_{1,reif}(a, b, v_1)$  and  $c_{2,reif}(a, b, v_2)$ . Since  $v_{12}$  has value *true*, and  $v_{12} = v_1 \wedge v_2$ ,  $v_1$  and  $v_2$  themselves must have the value *true*. Comparable reasoning can be used to infer values when *or* and *not* constraints are present. Although not strictly necessary, this information helps choosing non-reified constraints when not required.

It does enable another optimization though. When an equality constraint between two nodes exists, with a truth value that can be proven to be true using constant propagation, the constraint can be dropped and the nodes unified. An example will clarify the matter: the constraint model  $a = b + c$  over the three variables  $a$ ,  $b$  and  $c$  would be turned into a graph with 5 nodes ( $N_a$ ,  $N_b$ ,  $N_{a+b}$ ,  $N_c$  and  $N_{true}$ ), and 3 constraints:  $N_a + N_b = N_{a+b}$ ,  $(N_{a+b} = N_c) = N_{true}$  and  $N_{true} = true$ . In this case, the  $N_{a+b}$  and  $N_c$  nodes can be unified, which results in the simpler model with four nodes ( $N_a$ ,  $N_b$ ,  $N_c$ ,  $N_{true}$ ) and two constraints ( $N_a + N_b = N_c$ ) and  $N_{true} = true$ . Clearly the fourth node and the second constraint are redundant, simplifying the result to a single edge over three variables:  $N_a + N_b = N_c$ .

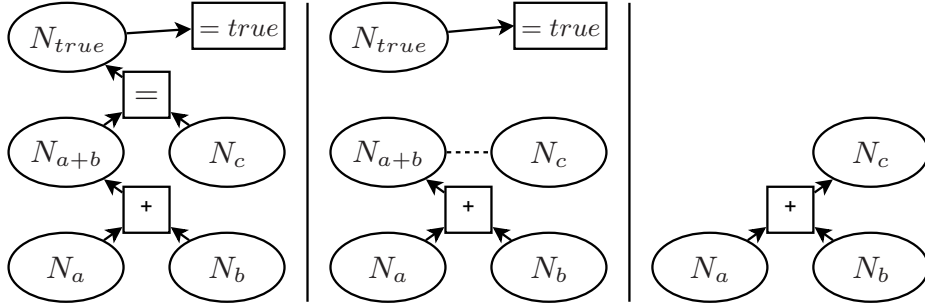


Fig. 2. Optimization: unification of equal nodes

### 2.3 Constraint Tiling

In the first phase of the compilation process, the high-level solver-independent model has been disassembled into a solver-independent constraint network graph for easier optimization. Now, in the last phase, the constraint graph is turned

back into a constraint model, but now of a solver-dependent and more low-level nature. The nodes and edges of the graph are mapped to constraint variables and constraints of the underlying constraint solver.

Our conversion algorithm is a *tiling* process, akin to instruction tiling in compiler backends, that matches subgraphs against “tiles” supported by the underlying solver. While we assume that small tiles are available for every type of node and edge in the graph, preference is given to larger tiles that cover multiple nodes and edges. Note that the main difference with traditional tiling is that we consider a graph structure rather than a tree structure. While this may somewhat complicate matters, there is potentially more information available to make good tiling decisions.

In a first step, we decide which nodes in the graph become the variables of the resulting constraint problem, and how other nodes can be written as a function of those, by “absorbing” edges into annotations on nodes. In a second step, these annotations are used together with the remaining edges to generate the final low-level constraints.

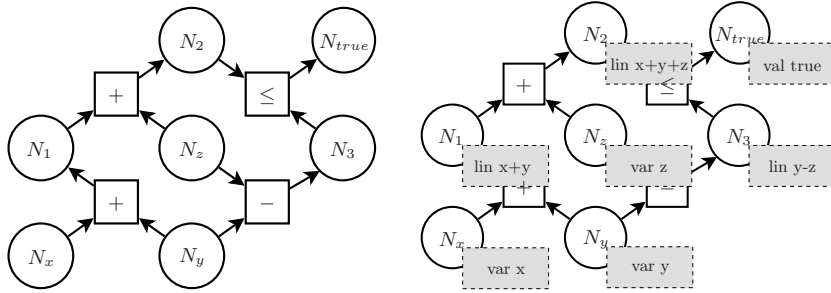
*Example 1.* Assume the following model expression:

$$x + y + z \leq z - y \quad (1)$$

Further assume the underlying solver supports linear inequalities:

$$\sum_{i=1}^n a_i v_i \leq c \quad (2)$$

where  $a_i$  and  $c$  are constants, and  $v_i$  are variables. Clearly, the given constraint (1) can be translated to a single linear inequality constraint, with  $\bar{a} = [1, 2, 0]$ ,  $\bar{v} = [x, y, z]$ ,  $c = 0$ , or even simpler, with  $\bar{a} = [1, 2]$ ,  $\bar{v} = [x, y]$ ,  $c = 0$ .



**Fig. 3.** Constraint Network Graph for  $x + y + z \leq z - y$ , unannotated and annotated

The calculated constraint network (see Figure 3) has an edge corresponding to the inequality, with nodes corresponding to  $x + y + z$  ( $N_2$ ) and  $z - y$  ( $N_3$ ).

Furthermore,  $N_2$  is connected using a  $+$  edge to the nodes corresponding to  $x + y$  ( $N_1$ ) and  $z$ . To be able to match this as a single linear inequality, it is necessary to know that all these nodes can be considered linear combinations of other nodes. We capture this information in a node annotation  $linear(\bar{a}, \bar{v}, c)$ , meaning:

$$linear(\bar{a}, \bar{v}, c) = c + \sum_{i=1}^n a_i v_i \quad (3)$$

Recognizing that  $N_2$  is connected using a *plus* edge to two other nodes, one can try to find out whether those two nodes can be considered linear combinations of other nodes as well (or simple constants or variables). Since one of them is again connected using a *plus* to  $N_1$ , this matching continues recursively while traversing the graph. Eventually, nodes are reached that can no longer be considered linear combinations of other (not yet explored) nodes. These end nodes become constraint variables, while all nodes on the paths from the inequality to the end nodes, are considered linear functions of others. All this information is materialized as an annotation on graph nodes, to avoid later recomputation. Finally, using the following formula, the inequality is turned into the obvious single linear inequality constraint:

$$linear(\bar{a}_1, \bar{v}_1, c_1) \leq linear(\bar{a}_2, \bar{v}_2, c_2) \Leftrightarrow \sum_{i=1}^n a_{1i} v_{1i} + \sum_{i=1}^m (-a_{2i}) v_{2i} \leq c_2 - c_1 \quad (4)$$

There are many more useful examples of annotations, including (potentially parametrized) constant values, known sizes of array variables, and certain structures imposed by global constraints (such as *alldifferent*).

To recognize the patterns, we build them all simultaneously. In fact, we want to create an annotation for every node in the graph, potentially using a trivial annotation that describes it as a simple variable. To do so, We put all potential annotations (called annotation generators) in a priority queue and process them one by one, starting from those that have a chance to create a “better” annotation (a chance for a constant value annotation is given highest priority, and creating a separate variable lowest priority). Annotation generators may depend on the presence of specific annotations on other nodes, in which case the corresponding generator may be called prematurely, and removed from the queue.

There is one further complication: not each type of annotation may be implemented or even be useful for every edge. For example, there is typically no constraint that implements a division between linear combinations of variables. Therefore the *division* edge should not support the linear annotation. Which annotations are possible and useful depends on the solver.

The resulting algorithm is as follows:

1. For each edge type, the solver interface is asked which annotation types it supports for its vertices, and which annotation generators are available.
2. Using a partial ordering on these generators, they are processed one by one:

- The generator is skipped if it would not produce any useful annotation. This is the case if it is for a node which is already fully specified (annotations compatible with all its edges have been created already).
  - The generator can request annotations (of a particular type) of neighboring nodes. This fails if a dependency loop occurs or no generator for that node/type combination exists. Otherwise the respective generator is invoked, its resulting annotation stored, and passed back to the calling generator.
  - Based on this information, the calling generator as a whole may fail, or create the annotation.
  - If this results in a second annotation for a same node, an artificial equality constraint between it and the earlier one is added.
3. If nodes without annotations remain, new constraint variables are created for them.
  4. Finally, the remaining edges and annotations for their vertices are passed to the solver interface to produce low-level constraints.

The result is an eager matching algorithm that consumes edges in the graph by describing some nodes in it in function of other nodes, possibly recursively. Nodes for which this is not possible become simple constraint variables, and edges for which this is not possible become real constraints.

### 3 Implementation

The techniques described in the previous section were used in a new finite domain layer for MCP [6]. It consists of:

- a general component for creating expression trees in Haskell, and performing simple optimizations on them
- a high-level modeling front-end, on top of MCP’s monad[8]-based technique for writing search trees
- a translation layer based on the previous section, exposing an MCP solver interface that uses high-level boolean expressions as constraints, delegating the real propagation and pruning to an underlying solver
- some instances of MCP solver interfaces that support interaction with the FD layer, including one Gecode[2] interface that is used by both runtime back-ends and code-generation back-ends

#### 3.1 Expression trees

The general expression component provides three basic expression types: booleans, integers and arrays of integers. We do not fix the type of variables these expressions can refer to yet, so they can be used independently. Without going into details, they are very conveniently represented by Haskell’s algebraic data types (ADTs).

Using Haskell’s support for defining new operators and overloading operators from certain predefined classes (such as `+`, `-` and `*`), it is possible to create a concise syntax for writing expression trees. Simple pattern-matching based simplification rules are applied on the trees while building them. For example, `5 + 3` evaluates to the constant 8, while `2*a-a+3` evaluates to  $a + 3$ .

Higher-order constructs are equally easy to represent. Since Haskell is a true functional language, it allows storing the inner functions for `map`, `fold` and `forall` as first-class elements in the expression tree.

Here is a full list of supported operators and functions:

- Overloaded operators: `+`, `-`, `*`
- Overloaded functions: `div`, `mod`, `abs`, `negate`, `succ`, `pred`
- Arithmetic operators: `@/`, `@%`
- Array operators: indexing (`!`, `@!!`), range (`@..`), concatenation (`@++`)
- Equality: `@=`, `@/=`
- Inequality: (`@/=`, `@<`, `@>`, `@>=`, `@<=`)
- Boolean: `@||`, `@&&`, `inv`, `channel`
- Conditional: boolean: `@?`, integer: `@??`
- Higher order: `cfold`, `cmap`, `slice`, `loopall`, `loopany`, `forall`, `forany`
- Array functions: `thead`, `ctail`, `list`, `size`, `csum`
- Global constraints: `sorted`, `allDiff`, `allin`

### 3.2 Modeling layer

The modeling layer provides a way for writing high-level constraints using a common FD syntax. As said, boolean expressions are used as constraints. To this end, we use the expression tree component described in the previous section, using unique identifiers as terms.

In MCP, a constraint model is simply an initial search tree, with unexplored nodes at the leaves. These nodes are created based on the branching strategy, and during search they are evaluated and replaced by subtrees.

A function is provided that turns a constraint for a particular solver into a minimal search tree that adds this constraint, called `add`. Without any further syntactic sugar, an initial model for the problem  $x > 5 \wedge x < 10 \wedge x^2 = 49$  would need to be written as:

```

1 model = exists $ \x -> do    -- request a variable x
2   add $ x @> 5                -- state that x>5
3   add $ x @< 10               -- state that x<10
4   add $ x*x @= 49             -- state that x*x=49
5   return x                    -- return x

```

To avoid the need for calling `add` for each constraint, alternative versions of the boolean operators are created that implicitly call the `add` function, instead of simply returning a boolean expression.

Without further modifications, this requires two different versions of all boolean operators and functions to exist: one simple expression form, and another as a model tree that posts the respective boolean expression as constraint.

It is however possible to turn model trees back into expressions, as long as no `Dynamic` nodes are used. This is exactly what we'll do: all operators and functions on the expression level that take a boolean expression, are masked behind a new operator or function that takes a model tree instead and turns it into an expression before passing it to the real expression operator or function. Likewise, all operators and functions that return a boolean expression are replaced by one that turns its result into a tree, using the `add` function.

```

1 {-# LANGUAGE TypeFamilies #-}
2
3 import Control.CP.FD.Example
4
5 -- diff: the differences between successive elements of an array
6 diff l = exists $ \d -> do      -- request an (array) variable d
7   let n = size l                -- introduce n as alias for size l
8       size d @= n-1            -- size of d must be one less than n
9   loopall (0,n-2) $ \i -> do  -- for each i in [0..n-2]
10     d!i @= abs (l!i - l!(i+1)) -- d[i] = abs(l[i]-l[i+1])
11   return d                     -- and return d to the caller
12
13 model :: ExampleModel ModelInt -- type signature
14 model n =                       -- function 'model' takes argument n
15   exists $ \x -> do            -- request an (array) variable x
16     size x @= n                -- whose size must be n
17     d <- diffList x           -- d becomes the "diff" of x
18     x 'allin' (0,n-1)         -- all x elements are in [0..n-1]
19     d 'allin' (1,n-1)         -- all d elements are in [1..n-1]
20     allDiff x                  -- all x elements are different
21     allDiff d                  -- all d elements are different
22     x @!! 0 @< x @!! 1        -- some symmetry breaking
23     d @!! 0 @> d ! (n-2)      -- some symmetry breaking
24     return x                   -- return the array itself
25
26 main = example_main_single_expr model

```

**Fig. 4.** The AllInterval problem in FD-MCP

Figure 4 shows a full Haskell program for the AllInterval problem. The aim is to find a sequence of numbers of size  $n$ , where each number is different, between 0 and  $n - 1$ , and the absolute values of differences between subsequent elements take all values between 1 and  $n - 1$ . Using monadic composition, we are able to abstract the creation of the difference array in a separate function (`diff`, lines 6–11), although it introduces an additional variable, and call it on line 17.

Note the use of `size 1` within the `diff` function. The implementation only supports arrays for which a (parametrized) size expression can be derived. Because of node unification, this is easy in this case: line 16 equates `is` to `n`.

### 3.3 FD layer

The actual FD layer integrates the high-level modeling layer described in the previous section with the MCP framework itself. Given an MCP solver that supports the FD interface, it provides a *wrapped* solver, which accepts the boolean expressions from the modeling layer as constraints instead of the constraints from the underlying solver. When constraints are posted to this wrapper-solver, they are accumulated until branching occurs, to take advantage of inter-constraint optimizations as described in Section 2.2. When this happens, all accumulated constraints are processed by an implementation of the system described in Section 2. Expressions are converted to their corresponding constraint network graph, node annotations are calculated, and remaining edges translated to constraints for the underlying solver.

As explained earlier, the graph representation can have edges that are labeled with a sub-graph, to represent higher order constraints without flattening them. Since typical solvers do not have any notion of loop constructs or arrays of indeterminate size, the FD layers provides default flatteners for all higher-order constructs, available as mixins.

### 3.4 Gecode interface

Although not the only back-ends, the Gecode-based solvers form the largest part of the implementation. Three different solvers exist:

- Two solvers that use the Gecode library at runtime through Haskell’s Foreign Function Interface
  - **RuntimeSolver**: uses MCP’s own search and branching, but sends posted constraints to C++, and retrieves variable domains from it
  - **SearchSolver**: delegates even search and branching to Gecode, by generating a search-tree in Haskell in which each node corresponds to searching one full solution in C++
- **CodegenSolver**: a pseudo-solver which does not actually do any propagation or pruning, but instead records all created variables and posted constraints. Afterwards, this information is used to generate a C++ program, that after compilation will use Gecode to search for solution, bypassing all of Haskell’s overhead at runtime.

Of particular importance here is the support for higher-order constraints. The Gecode layer’s constraint representation does support some higher-order constructs, to allow the code generation backend to create parametrized code. A side-effect is the ability to have higher-order class-level optimizations that are specific to the supported constraints. Three such optimizations are implemented

for the *fold* construct: one using an addition over a channeled equality test becomes a *count* constraint, a simple addition becomes a *sum*, and an addition over an arbitrary function of the input becomes a *sum* combined with a *map*.

## 4 Evaluation

To verify the quality of the translation, a set of classic CP problems were ported to FD-MCP, resulting in smaller code on average, and benchmarked. We compare the runtimes of original C++ Gecode implementations against runtimes of both our real Gecode solvers, as well as the runtime of the code generated by the C++ code generation pseudo-solver. Figure 5 compares runtimes<sup>1</sup> of six typical CP benchmarks for different problem sizes. The graphs clearly show that the generated C++ code has very close or even slightly better performance than the original benchmark. The C++ versions do some additional bookkeeping and have more options, resulting in slightly higher overhead sometimes. On the other hand, the generated code sometimes contains a small amount of superfluous variables, causing inefficiencies. When comparing with the direct solvers, larger differences occur. When using Gecode’s search, the overhead of switching from Haskell to C++ and back for each constraint can become significant, eg. in the *queens* benchmark. When using MCP’s search, this overhead also occurs during search, resulting in significantly lower performance. The benchmark results, as well as compilation times and measurements of lines of code<sup>2</sup>, are given in Appendix A.

## 5 Related work

Different languages and systems for CP modeling and solving exist:

Like FD-MCP, the Tailor[5] system attempts to present a high-level modeling interface to the user, with optimizations to prevent inefficient solving caused by users unfamiliar with the intricacies of CP. However, it is a separate tool that processes specific modeling languages (Essense’ and XCSP), and translates them to Minion, FlatZinc or Gecode programs. It lacks the flexibility MCP provides when solving directly, yet provides rather advanced model optimizations.

The Zinc[3] family of languages (including MiniZinc and FlatZinc) provide an extensive framework for translating from high-level models to flattened and solver-specific input. The translation is based on a rule-based system called ACD Term Rewriting[1], which performs transformations directly on the syntax tree. Zinc provides comparable higher-order constructs (only over arrays of known length), and converts boolean combinations of constraints to reified constraints.

---

<sup>1</sup> Benchmarks performed on a Intel Core 2 Duo E8500 system with 4GiB om RAM, running 64-bit Ubuntu 10.04, GHC 6.12.1, Gecode 3.2.1 and MCP 0.7.0. C++ benchmarks were modified to use the same search order as FD-MCP. Runtimes are averages over running each instance for 10 minutes.

<sup>2</sup> using sloccount, see <http://www.dwheeler.com/sloccount/>

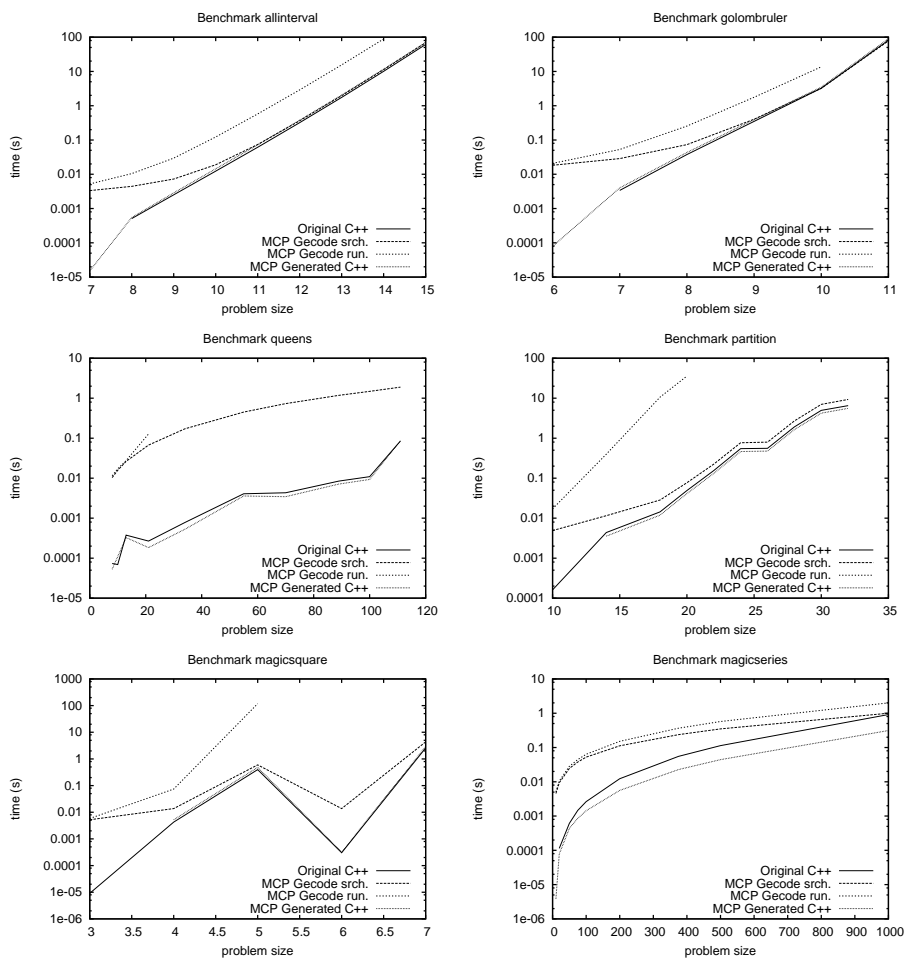


Fig. 5. Benchmarks

The multi-paradigm Mozart[7] programming system, based on the Oz language, includes support for constraint programming in a flexible and declarative way. Constraint solving concepts such as spaces are provided first-class in the system. It seems however mostly focused on the ability to program search and propagation, and lacks high-level modeling features such as writing constraints as expressions.

## 6 Conclusion and future work

We have shown how to extend FD-MCP with support for expression-based constraints and first-class higher order constructs. Furthermore a compilation scheme was designed that maps these high-level models to low-level solver-specific ones. These ideas were implemented and benchmarks show that the resulting performance is often comparable to native C++ Gecode benchmarks.

Future work includes extending the system with more specific optimizations, more global constraints and additional data types. Having created an abstraction for (FD) constraint modeling that supports compilation to C++ code in addition to direct solving, the same should be done for the branching and searching parts of CP. First by providing the ability to choose a search strategy from a predefined list, later by generating C++ code based on a specification in MCP.

## References

1. G. J. Duck, P. J. Stuckey, and S. Brand. ACD term rewriting. In S. Etalle and M. Truszczynski, editors, *ICLP*, volume 4079 of *LNCS*, pages 117–131, 2006.
2. Gecode Team. Gecode: Generic constraint development environment, 2006. Available from <http://www.gecode.org>.
3. K. Marriott et al. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.
4. S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.
5. A. Rendl. *Effective Compilation of Constraint Models*. PhD thesis, University of St. Andrews, January 2010. <http://www.cs.st-andrews.ac.uk/~andrea/>.
6. T. Schrijvers, P. Stuckey, and P. Wadler. Monadic constraint programming. *Journal of Functional Programming*, 2009.
7. M. Team. The mozart programming system, 2004. Available from <http://www.mozart-oz.org/>.
8. P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52, London, UK, 1995.
9. P. Wuille and T. Schrijvers. Monadic Constraint Programming with Gecode. In *Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation*, pages 171–185, 2009.
10. P. Wuille and T. Schrijvers. Parametrized models for on-line and off-line use. In *Preliminary proceedings of the 19th International Workshop on Functional and (Constraint) Logic Programming*, pages 65–79, 2010.

## A Benchmark results

name	size	Runtime				Compile time			Lines of code		
		C++	MCP		run	GCC		GHC Haskell	C++		Haskell
			gen.	search		C++	gen.		orig.	gen.	
allinterval	7	0.004	0.004	0.0073	0.0092	1.3	0.85	0.032	52	81	21
	8	0.0045	0.0046	0.0084	0.014						
	9	0.0065	0.0069	0.011	0.034						
	10	0.016	0.018	0.023	0.13						
	11	0.065	0.076	0.078	0.58						
	12	0.32	0.38	0.37	2.9						
	13	1.8	2.1	2	16						
	14	10	12	11	89						
	15	61	70	69	-						
bibd	0	0.0041	0.0053	0.086	0.094	1.3	0.86	0.042	119	113	19
efpa	0	0.0043	0.005	0.059	0.063	1.8	0.93	0.042	199	146	18
golombruler	6	0.0041	0.0041	0.022	0.025	1.3	0.86	0.042	87	94	26
	7	0.0074	0.0081	0.033	0.057						
	8	0.042	0.047	0.077	0.26						
	9	0.35	0.4	0.41	1.8						
	10	3.2	3.5	3.2	14						
	11	81	90	78	-						
grocery	0	0.099	0.099	0.093	0.097	1.3	0.84	0.029	42	69	11
magicseries	10	0.0039	0.0039	0.0085	0.009	1.3	0.85	0.032	62	98	14
	20	0.004	0.004	0.013	0.014						
	50	0.0045	0.0043	0.028	0.032						
	75	0.0054	0.0047	0.042	0.048						
	100	0.0065	0.0053	0.056	0.067						
	200	0.016	0.0095	0.12	0.16						
	375	0.059	0.027	0.24	0.37						
	500	0.12	0.048	0.35	0.58						
	1000	0.92	0.32	1	2						
magicsquare	3	0.0039	0.0039	0.0091	0.0097	1.4	0.87	0.042	62	87	26
	4	0.0081	0.0091	0.018	0.078						
	5	0.4	0.49	0.6	120						
	6	0.0042	0.0042	0.018	-						
	7	2.6	2.9	4.4	-						
partition	10	0.0044	0.0043	0.0092	0.021	1.4	0.89	0.042	74	97	27
	14	0.0087	0.0078	0.016	0.4						
	18	0.019	0.016	0.033	11						
	20	0.053	0.045	0.081	36						
	22	0.16	0.14	0.23	-						
	24	0.55	0.47	0.77	-						
	26	0.56	0.48	0.8	-						
	28	1.9	1.6	2.7	-						
	30	5	4.2	7	-						
		32	6.5	5.6	9.3	-					
queens	8	0.0039	0.0039	0.014	0.015	1.3	0.83	0.032	80	73	13
	10	0.0039	0.0039	0.02	0.021						
	13	0.0042	0.0042	0.03	0.031						
	21	0.0041	0.004	0.071	0.13						
	34	0.0046	0.0044	0.18	-						
	55	0.0079	0.0075	0.45	-						
	70	0.0082	0.0073	0.74	-						
	89	0.012	0.011	1.2	-						
	100	0.015	0.013	1.5	-						
	111	0.089	0.089	1.9	-						