Parameterized Models for On-line and Off-line Use

Pieter Wuille and Tom Schrijvers*

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

Abstract. The Monadic Constraint Programming framework leverages Haskell's rich static type system and powerful abstraction mechanisms to implement an embedded domain specific language (EDSL) for constraint programming.
In this paper we show how the same constraint model expressed in the EDSL can be processed in various modes by external constraint solvers. We distinguish between *on-line* and *off-line* use of solvers. In off-line mode, the model is not solved; instead it is compiled to lower-level code that will search for solutions when compiled and run. For on-line use, the search can be handled by either the framework or in the external solver. Off-line mode requires recompilation after each change to the model. To avoid repeated recompilation, we separate model from data by means of parameters that need not be known at compile time. Parametrization poses several challenges, which we resolve by embedding the EDSL more deeply.

1 Introduction

The Monadic Constraint Programming framework integrates constraint programming in the functional programming language Haskell [9] as a deeply embedded domain specific language (EDSL). This has a considerable advantage compared to special-purpose Functional Constraint (Logic) Programming (FCP) languages such as Curry [7] or TOY [5]. We directly obtain state-of-the-art functional programming support with zero effort, allowing us to focus on constraint programming itself.

While the integration is not as tight, Haskell does offer good EDSL support to make the embedding quite convenient. Moreover, being less tight does provide for greater flexibility. Aspects that are baked into some FCP languages, such as search strategies or the particular solver used, are much more easily interchanged from within the program. In addition, the deep embedding of the EDSL allows us to use the constraint model for more than straight (on-line) solving. For instance, transformations can be applied to the model for optimization purposes or to better target a particular constraint solver. Alternatively, the model does not have to be solved on-line, but can drive a code generator that produces an executable for off-line solving.

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

This paper reports on the FD-MCP module of the framework, specific to finite domain (FD) solvers. We show how the framework supports different modes of processing an FD model, by both on-line and off-line solvers. Then we identify the need for parametrized models to make the off-line solver approach both more useful and more efficient. We show how the framework is adjusted to support parametrized models, including deeply embedded iteration constructs and indexed lists of constraint variables.

2 Monadic Constraint Programming

The MCP [10] framework is a highly generic constraint programming framework for Haskell. It provides abstractions for writing constraint models, constraint solvers and search strategies. This paper focuses on the solving and modeling parts.

2.1 Generic Constraint Programming Infrastructure

MCP defines type classes, Haskell's form of interfaces, for `Solver`s and `Term`s:

```
class Monad s => Solver s where
  type Constraint s :: *
  type Label s :: *
  add :: Constraint s -> s Bool
  run :: s a -> a
  mark :: s (Label s)
  goto :: Label s -> s ()

class Solver s => Term s t where
  newvar :: s t
```

A type that implements the `Solver` type class must provide a type[1] to represent its constraints and labels, an `add` function for adding constraints, a `run` function to extract the results, a `mark` function to create a label of its current state, and a `goto` function to return to a previous state.

A solver type s must also be a monad [11]. A monadic value `s a` is an abstraction of a form of computation `s` that yields a result `a`. Constraint solvers are typically computations that thread an implicit state: the constraint store.

A solver also provides one or more types of terms: `Term s t` expresses that `t` is a term type of solver type `s`. Each term type provides a method `newvar` to generate new constraint variables of that type.

MCP also defines a data type `Model`, representing a model tree:

```
data Model s a
 = Return a                              -- return a value
 | Add (Constraint s) (Model s a)        -- add a constraint
```

[1] Implemented using associated types in Haskell

```
| NewVar (t -> Model s a) (Model s a)  -- introduce variable
| Try (Model s a) (Model s a)          -- disjunction
  ...
```

The model tree is parametrized in the constraint solver `s` and returned result type `a`. This provides a type-safe way for representing constraint problem models for arbitrary solvers and result types. The `Return` node returns solutions. The `NewVar` node introduces new variables. It takes a function as argument that produces the subtree when passed the new variable. This encoding prevents using a variable outside of the part of the tree where it is defined.

On top of the model data type, MCP provides syntactic sugar (functions that construct model trees), such as `exists` (create a variable), `exist n` (create a list of `n` variables), `addC` (add a constraint), `/\` (conjunction), `\/` (disjunction), `conj` (conjunction of list of models), ... Finally, `Model s` is also a monad.

2.2 The FD-MCP Module

The FD-MCP framework introduces an extra layer of abstraction between the more generic `Solver` interface of the MCP framework and the concrete solver implementations.

In contrast to MCP's generic `Solver` interface, which is parametric in the constraint domain, the `FDSolver` interface of FD-MCP is fully aware of the finite domain (FD) constraint domain: both its syntax (terms and constraints) and meaning (constraint theory). It does however make abstraction of the particular FD solver and e.g., propagation techniques used. Hence, it provides a uniform modeling language that abstracts from the syntactic differences between various FD solvers.

On the one hand, this allows the development of solver-independent models, model transformations (e.g., for optimization) and model abstractions (capturing frequently used patterns). On the other hand, specific solvers may focus on the efficient processing of their constraint primitives without worrying about modeling infrastructure.

FD-MCP Modeling Primitives The FD-MCP modeling language is built as a wrapper on top of the MCP solver interface. This way, the domain-independent combinators of the MCP framework, such as conjunction `(/\)` and existential quantification `exists` are available for FD models. The FD-MCP modeling language adds FD-specific constructs to that. Advanced FD constructs are defined in terms of a set of core primitives, resulting in a layered structure.

The FD layer abstracts from the solver-level concepts of variables and constraints, and allows everything to be expressed as arbitrarily-nested expressions. Boolean expressions are used as constraints, implicitly supporting reification.

To improve genericity, the used expression data types are parametrized in the type of terms they refer to. `i` refers to the type of integers terms, `b` to the type of boolean terms.

```
data BoolExpr i b
 = BoolVar b                                  [[BooolVar v]]  ≡  [[v]]
 | BoolConst Bool                             [[BoolConst n]] ≡  n
 | BoolAnd (BoolExpr i b) (BoolExpr i b)      [[BoolAnd a b]] ≡  [[a]] ∧ [[b]]
 | ...
 | Same (IntExpr i b) (IntExpr i b)           [[Same x y]]    ≡  [[x]] = [[y]]
 | Less (IntExpr i b) (IntExpr i b)           [[Less x y]]    ≡  [[x]] < [[y]]
 | Dom (IntExpr i b) (Integer,Integer)        [[Dom x y z]]   ≡  [[x]] ∈ {y,...,z}
 | AllDiff (ListExpr i b)      [[AllDiff [x1,...,xn]]] ≡  ⋀_{i≠j} x_i ≠ x_j
 | ...

data IntExpr i b
 = IntVar i                                   [[IntVar v]]    ≡  [[v]]
 | IntConst Integer                           [[IntConst n]]  ≡  n
 | Plus (IntExpr i l b) (IntExpr i l b)       [[Plus x y]]    ≡  [[x]] + [[y]]
 | Mult (IntExpr i l b) (IntExpr i l b)       [[Mult x y]]    ≡  [[x]] * [[y]]
 | ...
```

where the comment after each constructor shows its denotation. In the specific instance of using them as expressions, `IntExpr (IntTerm s) (BoolTerm s)` and `BoolExpr (IntTerm s) (BoolTerm s)` will be used. The types `IntTerm s` and `BoolTerm s` will be explained in Section 2.2.

Syntactic sugar On top of the core primitives, a number of convenient abstractions and syntactic sugar exists. Firstly, standard arithmetic operators and integer literals can be used for `IntExpr` thanks to an implementation of Haskell's `Num` type class.

```
instance Num (IntExpr i b) where
  fromInteger = IntConst
  (+)         = Plus
  ...
```

Thus `Plus x (Mult (IntConst 2) y)` can be written succinctly as `x + 2 * y`.

Furthermore — to simplify writing models — syntactic sugar exists that adds specific boolean expressions immediately as constraints in a model: [2]

```
x @<  y      = Add (Less x y) true
x @>  y      = y @< x
x @>= y      = x + 1 @> y
x @: (l,u)   = Dom x l u
xs `allin` d = conj [ x @: d | x <- xs ]
...
```

[2] The `[ x @: d | x <- xs ]` expression is a list comprehension, meaning "a list of `x @: d`, for each `x` from the list `xs`".

```
1 model = exist 4 $ \list@[a,b,c,d] -> do
2   list `allin` (0,711)
3   a + b + c + d @= 711
4   a * b * c * d @= 711000000
5   sorted list
6   return list
```

Fig. 1. 7-11 puzzle example in FD-MCP

In practice, these operators do not naively build an expression, but perform simplifications as well. For example:

```
x @<  y  = Add (Less (simplify x) (simplify y)) true

simplify (Plus x (IntConst 0)) = x
simplify (Mult x (IntConst 0)) = IntConst 0
simplify ...                   = ...
```

i.e. `@<` acts as a smart constructor that first simplifies its arguments. The simplifications are generic transformations, and form a first optimization step. Independent from the solver, they apply identities such as $x + 0 = x$ and $0x = 0$ (shown above) to the model.

Example Figure 1 illustrates the resulting FD-MCP modeling language with a model for the classical 7-11 puzzle. This puzzle concerns 4 unknown amounts whose sum and product are both 7.11. The amounts are represented in units of 1 cent. [3]

Line 1 starts the declaration of an FD-MCP model tree `model`. `exist` requests a new list of 4 variables and has an (anonymous) callback function for handling them as argument. This function itself takes the `list` of new variables as argument, individually aliased as `a`, `b`, `c` and `d`. Lines 2–5 define a conjunction of constraints that must hold for these variables:

- Line 2: restrict all variables' domains to $0 \dots 711$.
- Line 3: the sum of the four variables is 711.
- Line 4: the product of the four variables is 711000000.
- Line 5: each variable is larger than or equal to the next one (to break symmetry).

Mapping to the solver backend The conversion of high-level solver-independent expressions to low-level solver-dependent constraints and variables is a complex

[3] The `$` operator is function application (`f $ x = f x`) which associates to the right instead of the left. For instance, `f $ g $ h x` is equivalent to `f (g (h x))`, but syntactically more convenient.

operation. Often there will be different possible outcomes, with varying performance characteristics for solving. Hence it is a good candidate for optimization.

The FD-MCP system uses a graph-based compilation scheme to do this conversion. Details of this compilation scheme are outside of the scope of this paper; the explanation can be found in [13].

To have a solver integrated with FD-MCP, it must — in addition to the `Solver` class — also instantiate the `FDSolver` class:

```
class (Term s (IntTerm s), Term s (BoolTerm s)) => FDSolver s where
  type IntTerm s :: *
  type BoolTerm s :: *
  ...
  enforce :: BoolExpr (IntTerm s) (BoolTerm s) -> s Bool
```

This defines the `FDSolver s` class, which demands implementors to specify the associated types `IntTerm s` and `BoolTerm s`. These are the types of variables that will be used in the low-level form. In what follows, we will assume the class also defines a function `enforce` that maps a boolean expression over its terms to a monadic action that enforces the truth of that expression in its constraint store, returning `False` if this leads to insatisfiability. In reality, the class is more extensive, and does not offload all responsibility to the solver.

Integration with MCP The `FDSolver` type class allows us to define a generic solver `FDWrapper s` that encapsulates the mapping from the generic model to the solver-specific model.

The `FDWrapper s` is an MCP `Solver` which uses `BoolExpr (IntTerm s) (BoolTerm s)` as constraints and `BoolExpr`'s and `IntExpr`'s as terms.[4]

```
1  newtype FDWrapper s a = FDWrapper { unFD :: StateT (FDState s) s a }
2  data FDState s = FDState { cons :: BoolExpr (IntTerm s) (BoolTerm s) }
3
4  commit :: FDSolver s => FDWrapper s ()
5  commit = do
6    s <- get
7    FDInstance $ lift $ enforce $ constraint s
8    put s { constraint = BoolConst True }
9
10 instance FDSolver s => Solver (FDWrapper s) where
11   type Constraint (FDWrapper s) = BoolExpr (IntTerm s) (BoolTerm s)
12   type Label (FDWrapper s) = Label s
13   add c = do
14     s <- get
15     put s { cons = (cons s) @&& c }
16   mark = do
17     commit
```

[4] The instance requires `s` to belong to the `FDSolver` class, which requires a type `IntTerm s` to belong to class `Term s`, which requires `s` to belong to class `Solver` itself.

```
18      FDInstance $ lift $ mark
19    goto l = do
20      put initFDState
21      FDInstance $ lift $ goto l
22    run f = run $ evalStateT (unFD f) initFDState
23
24  instance Term (FDWrapper s) (IntExpr (IntTerm s) (BoolTerm s)) where
25    newvar = do
26      nv <- FDInstance $ lift $ newvar
27      return $ IntVar nv
```

Line 1 defines the `FDWrapper s a` type, a wrapper around a `s` solver, transformed using the `StateT` monad transformer to maintain an additional state (`FDState s`) through the computation. This state — defined on line 2 — contains a boolean expression `cons` that represents the conjunction of constraints that are not yet applied to the underlying constraint solver. Lines 4–8 define the function `commit`, which flushes this conjunction to the underlying solver using the earlier `enforce` function, and `lift`, which transforms an action for the `s` monad to one for the `FDWrapper s` monad. Finally, `FDWrapper s` is made an instance of `Solver` itself in lines 10–22. Line 11 states that boolean expressions will be acceptable as constraints, and line 12 declares that labels of the underlying solver are reused for the wrapper. The `add` function on lines 13–15 simply adds constraints to the conjunction (using the boolean expression operator `@&&`). The `mark` and `goto` functions will flush the constraints, and call the respective function of the underlying solver. Running the wrapped solver (line 22) requires evaluating it using an initial state, and then running the resulting monad action in the underlying solver. Expressions are declared as terms for `FDWrapper s` (lines 24–27) by wrapping new underlying variables in a `IntVar` constructor to form `IntExpr`'s.

3 Solver Backends and Modes

The initial release of the MCP framework featured only one solver, a simple FD solver implemented in Haskell. However, rather than implement a solver in Haskell, it is much more attractive to interface external state-of-the-art solvers implemented in lower-level languages. That is why we have provided an interface to the Gecode FD solver in C++ [12]. In this work we expand considerably upon this initial interface and show how the same external solver can be interfaced in different modes.

3.1 On-line and Off-line Modes

Firstly, we distinguish between *on-line* and *off-line* use. The former means that the constraint model is processed by the MCP framework, in collaboration with the solver, to produce solutions. This mode is used for the original Haskell-based FD solver. The latter concerns *staged compilation*: in the first stage, the FD model is processed by the MCP framework that produces code for the second stage in the solver's programming language; the stage-2 code produces solutions. This mode was used in the original Gecode backend of [12]. The off-line mode comes with a compilation function $\langle\!| \cdot |\!\rangle$ :: `Model OfflineSolver` $a \rightarrow$ `C++` instead of the usual `run` function for solvers.

The off-line mode has a clear appeal for performance reasons: it avoids the interpretative overhead when solving the constraint model in the second stage. Of course,

there is the compilation overhead of the first stage. We come back to this issue in the next section, where we considerably improve the usefulness of the off-line mode.

The on-line mode is very convenient for programming the search: all the high-level search features of the MCP framework are available. In contrast, our off-line Gecode solver provides a fixed search strategy. A considerable disadvantage of the on-line mode is the interpretative overhead of Haskell, which is confounded by the fact that the FD solver is implemented in Haskell itself.

New on-line Gecode solver In this paper we present a new on-line mode for the external Gecode solver. This combines the performance of Gecode with the high-level search features of the MCP framework. The solver type is defined as:

```
newtype OnlineSolver a
  = OnlineSolver { runOnline :: StateT GecodeState IO a }
```

The `OnlineSolver` is a monad composition of:

- **the `IO` monad:** to access the Gecode library through the Haskell Foreign Function Interface (FFI), and
- **the `StateT GecodeState` monad transformer:** to maintain the solver state:

  ```
  data GecodeState = GecodeState { space :: Space
                                 , cexpr :: Map IntExpr IntTerm }
  ```

  which consists of a reference to the current Gecode space, and a map to translate FD expressions in the constraint model to constraint variables in the Gecode solver.

The `OnlineSolver` is recognized as an actual solver by the framework with the following instance:

```
instance Solver OnlineSolver where
   type Constraint OnlineSolver =  GecodeConstraint
   add c  =  addOnlineGecode c
   run m  =  unsafePerformIO $ do state <- newState
                                  evalStateT state (runOnline m)
   type Label OnlineSolver      =  GecodeState
   mark   =  get
   goto s =  copyState s >>= put
```

The supported constraints of this solver are of type `GecodeConstraint`. The `addOnlineGecode` function adds a contraint to the current Gecode space, through the FFI. This involves constructing the constraint arguments, the FD expressions, in the Gecode solver. The `cexpr` helps out here, capturing earlier mappings of constraint variables and other FD expressions that already have a representation in the Gecode solver. This results in dynamic common subexpression elimination. Running the solver means running the underlying `IO` monad and the state transformer, with appropriate initial state.

Finally, for disjunctive models and branches in the search tree, we use the copying technique in Gecode. Thus for the label of a solver state, we simply use the solver state, i.e. the Gecode space, itself. Whenever creating a branch starting from a given space, we install a copy of that space as the current space so as not to affect other branches.

Thanks to this relatively simple instance, we can now use the MCP infrastructure (e.g., a search queue, compositional search transformers and enumeration) for the on-line Gecode solver.

3.2 Programmed versus Fixed Search Modes

The new on-line Gecode backend of the framework offloads constraint propagation on the Gecode solver, but still allows the programmer to program and specify the search heuristics through the high-level interface. We call this approach the *programmed search mode*. It has clear advantages in terms of expressivity, but it does incur an interpretative penalty for search, which for many constraint problems has a considerable impact on the overall solving time.

In order to avoid the interpretative overhead for search, we provide a second mode of on-line use, the *fixed search mode*. Just like the off-line Gecode solver, this mode provides a fixed search strategy implemented in C++ for the on-line Gecode solver. In this mode, labelling the model does not produce a whole subtree that is affected by the framework's search heuristics. Instead, a single node is generated on the MCP side that corresponds to many nodes in the Gecode solver which are processed by a fixed search strategy.

4 Parameterized Models

Many FD models are naturally parameterized in a problem size and/or other instance-specific integer values. For instance, the n-queens problem is parameterized in the board size, the Golomb ruler problem is parameterized in the ruler size, ...

Such parameterization does not pose any problem for the on-line solvers. The parameterized model is simply written as a *model function* from one (or more) integer value to an FD model. An `FDModel` is simply a `Model` for an `FDSolver`, that returns a list of solutions.

```
pmodel :: Int -> FDModel s
pmodel = \n -> ...
```

In order to solve the model, the model function is applied to the appropriate values, and the resulting model is handed to the on-line solver. No surprises.

For off-line solvers, we could follow the same technique. However, then we would obtain a non-parameterized off-line executable. Each time we would like to change the parameters, we would have to generate a new off-line executable! That is very costly in terms of compilation times, compared to the on-line solvers. The latter require only one invocation of the Haskell compiler for a parameterized model, while the former requires one invocation of the Haskell compiler and subsequently, for each instantiation of the parameters, an invocation of the C++ compiler. Moreover, the size of the off-line code is dependent on the problem size, because the framework fully flattens the model before generating code. Hence, the larger the problem size, the bigger the generated C++ code, and the longer the C++ compilation times. In summary, a new approach is necessary to make parameterized models practical for off-line solving.

The remainder of this section shows our approach for representing and compiling parameterized models. It has the two desirable properties: 1) a parameterized model requires only a single invocation of the C++ compiler, and 2) the generated code does not depend on the parameter value.

4.1 Parameters

We still represent parameterized models by model functions, but the functions take expressions rather than integers as arguments.

```
pmodel :: IntExpr (IntTerm s) (BoolTerm s) -> FDModel s
pmodel = \n -> ...
```

For brevity, we will omit the type parameters `(IntTerm s)` and `(BoolTerm s)` in further signatures mentioning FD expressions and models.

We still retain the above functionality for off-line solvers, as integer values can be lifted to FD expressions using the `IntConst :: Integer -> IntExpr` constructor. Moreover, `IntExpr i b` is also an instance of the `Num` type class, so integer literals can be supplied directly as arguments: `pmodel 1425`.

Of more interest is of course the treatment of model functions for off-line solving. A model function is compiled by applying it to special `IntExpr` values that represent *deferred values* . These deferred values will not be known until the C++ stage. We denote a deferred value in the first stage as `` `p ``, where p is the corresponding representation, a C++ `int` variable, in the second stage.

So using these deferred variables, we again obtain an `FDModel` that can be compiled much as before. Only the deferred values require special care. They are mapped to `int` instance variables of the generated C++ class that represents the Gecode constraint model. A new instance of the problem is created by instantiating an object of that class with the desired integer values for the parameters.

4.2 Indexed Constraint Variable Lists

Unfortunately, this is not the end of the story. Parameters of type `IntExpr` have fewer uses than values of type `Integer`. Indeed, the former can be used as arguments to constraints, but the latter can appear in many useful Haskell library functions as well as several functions of the MCP framework. Perhaps the most essential such function is `exist :: Integer -> ([t] -> FDModel) -> FDModel`, which creates a list of the specified number of constraint variables. In many parametric models, the number of constraint variables depends on the parameter value.

However, for the off-line solver, the integer value of the parameter is not available. Thus the actual creation of the list must be deferred from the on-line Haskell phase to the off-line phase. Moreover, we may wish to use a different data structure than a linked list in the off-line phase, such as an array in C++.

Hence, to allow writing models that can be used with both on-line and off-line solvers, we extend the expression component with explicit list expressions:

```
data ListExpr i b l
 = List [IntExpr i b l]
 | ListVar l
 | ListCat (ListExpr i b l) (ListExpr i b l)

data IntExpr i b l
 = ...
 | ListSize (ListExpr i b l)
 | ListAt (ListExpr i b l) (IntExpr i b l)
```

```
data BoolExpr i b l
 = ...
 | ListEqual (ListExpr i b l) (ListExpr i b l)
```

So an additional type parameter `l` is added to the definitions of `IntExpr` and `BoolExpr`, and a new expression type `ListExpr` is added. Furthermore, the `FDSolver` class is extended with a `ListTerm s` associated type.

For on-line solvers like `OvertonFD`, it is defined as an on-line Haskell list:

```
type instance ListTerm OvertonFD = [IntTerm OvertonFD]
```

For off-line solvers like `OfflineGecode`, a deferred list $`c$ is used that only records an identifier c of the particular list:

```
type instance ListTerm OfflineGecode = OfflineList Int
```

However, when writing a constraint model that is polymorphic in the solver type, `ListTerm` acts as an abstract data type that only allows a limited number of operations, supported by both on-line and off-line solvers. Typically, all interaction with lists is done using the higher-level `ListExpr` type. Supported functions on these expressions include:

- `fdexist :: IntExpr -> (ListExpr -> FDModel) -> FDModel`
  creates a new list of specified size, and acts as a generalization of `exist`. This function is implemented in terms of `exist` for on-line solvers, but creates a new deferred list for off-line solvers. Note that the size of generated code for the latter is constant (a single array declaration) as opposed to linear like `exist`.
- `(!) :: ListExpr -> IntExpr -> IntExpr` returns an element at a given index in the list. For on-line solvers it is implemented in terms of list indexation `(!!)`, but for off-line solvers a term denoting deferred indexation is returned. Then we have that $\langle\!\langle `c \mathbin{!} i \rangle\!\rangle = c[\langle\!\langle i \rangle\!\rangle]$.
- `collect :: [IntExpr] -> ListExpr` turns a Haskell list of variables into a list expression.

Global constraints form another class of functions that involve lists. These have been modified to support list expressions instead of Haskell lists:

- `allDiff :: ListExpr -> FDModel` all expressions in the given list expression evaluate to mutually distinct values.
- `sorted :: ListExpr -> FDModel` the given list expression is sorted.
- `allin :: ListExpr -> (IntExpr,IntExpr) -> FDModel`
  all expressions in the given list expression have a value between the given lower and upper bounds.

4.3 Iteration

Often the above operations for lists are not expressive enough. Instead of imposing global constraints on a list or indexing specific entries, many models process all elements of a list one at a time. For this purpose an iteration construct is necessary.

Iteration Primitives We introduce in our framework the iteration primitive `foreach :: (IntExpr,IntExpr) -> (IntExpr -> FDModel) -> FDModel`, whose denotation is:

$$[\![\texttt{foreach}\ (l,u)\ f]\!] \equiv \bigwedge_{i=l}^{u} [\![f\ i]\!]$$

For instance, we write $\bigwedge_{i=1}^{n}(c_i > i)$ as:

```
foreach (1,n) $ \i -> (c ! i) @> i
```

This is implemented by again extending the expression component:

```
data BoolExpr i b l
 = ...
 | ForEach (IntExpr i b l,IntExpr i b l) (IntExpr -> BoolExpr)
```

This time, we use the fact that functions are first-class objects in Haskell, and store the inner expression as a function in the outer expression. This immediately adds support for nesting and reification.

For on-line solvers, `ForEach` is translated literally according to its semantics:

```
enforce (ForEach (IntConst l,IntConst u) f)  = conj [ f i | i <- [l..u] ]
```

However, for off-line solvers, the range of the loop may not be constant, if it depends on a model parameter. Even if we do know the range, we may choose not to flatten the loop if the range is too large. In these cases, `foreach` is compiled to a C++ `for`-loop:

$$\langle\!\langle \texttt{ForEach}\ (l,u)\ f \rangle\!\rangle = \texttt{for (int i = } \langle\!\langle l \rangle\!\rangle \texttt{; i =< } \langle\!\langle u \rangle\!\rangle \texttt{; i++) \{ } \langle\!\langle f\ \texttt{`i} \rangle\!\rangle \texttt{ \}}$$

So the size of the generated code does not depend on the size of the iteration range.

Because iteration over the whole range, rather than a subrange, of a list occurs quite frequently, we introduce a second iteration construct `forall :: ListExpr -> (IntExpr -> FDModel) -> FDModel`, whose denotation is:

$$[\![\texttt{forall}\ c\ f]\!] \equiv \bigwedge_{v \in c} [\![f\ c]\!]$$

For instance, we write $\bigwedge_{v \in c}(v > i)$ as:

```
forall c $ \v -> v @> i
```

or even shorter:

```
forall c (@> i)
```

`forall` is simply mapped to the more generic `foreach` construct:

```
forall c f = foreach (IntConst 1, ListSize c) $ \i -> f (c ! i)
```

which means that we get the following C++ code:

$$\langle\!\langle \texttt{forall `}c\ f \rangle\!\rangle = \texttt{for (int i = 0; i < } \langle\!\langle \texttt{size `}c \rangle\!\rangle \texttt{; i++) \{ } \langle\!\langle f\ \texttt{`}c\texttt{[i]} \rangle\!\rangle \texttt{ \}}$$

Example The following program is a parameterized model of the n-queens problem:

```
nqueens n =                          --  define model function 'nqueens'
  fdexist n $ \q -> do               --  new list 'q' of size n
    q `allin` (1,n)                  --  all variables in range [1..n]
    foreach (1,n) $ \i ->            --  for i in [1..n]
      foreach (i+1,n) $ \j  -> do    --  for j in [i+1..n]
        q!i     @/= q!j              --  \
        q!i + i @/= q!j + j          --  | constraints
        q!i - i @/= q!j - j          --  /
    return q                         --  return result list
```

Except for import statements and a main function that inputs the parameter value, calls the solver and outputs results, this is a fully working Haskell program.

Additional Higher-Order Constructs While the above iteration primitives are expressive enough to formulate most list processing operations, often the formulation can be quite awkward and the resulting code rather inefficient (e.g. requiring auxiliary lists). For that reason, we directly support additional higher-order list processing constructs besides `forall` and `foreach`.

Mimicking the standard Haskell functions `map` and `foldl`, we provide the following:

- `fdmap :: (IntExpr -> IntExpr) -> ListExpr -> ListExpr` transforms each element of a list using a specified function, similar to the standard Haskell function `map`.
- `fdfold :: (IntExpr -> IntExpr -> BoolExpr) -> IntExpr -> ListExpr -> IntExpr` folds a list to a single expression, similar to the standard Haskell function `foldl`.

Again, the `IntExpr` and `ListExpr` types are extended with new constructors (`Map` and `Fold`), and simplifications are applied when creating them. For example, applying a `fdmap` on another `fdmap` results in a single `Map`, while applying an `fdfold` on an `fdmap` will result in a single `Fold`.

In the case of a Gecode-based solver (either on-line or off-line), further optimizations are applied. For example, a `Fold` that reduces a list using the `(+)` operator will be translated to a single efficient sum constraint propagator.

5 Evaluation

In order to evaluate the two new extensions, the on-line Gecode solver support and the support for parameterized models, existing benchmarks for Gecode have been ported to FD-MCP. Tables 1, 2 and 3 show the results. Lines of code (LoC) are measured using SLOCcount[5], the timings in seconds are average CPU times over multiple runs.[6]

[5] `http://www.dwheeler.com/sloccount/`

[6] Benchmarks have been performed on a 64-bit Ubuntu 10.04 system using a 3.16GHz Intel® Core™2 Duo E8500 processor, with 4GiB RAM. Software versions: GHC 6.12.1, GNU G++ 4.4.3, Gecode 3.2.1.

5.1 Solving Results

The first table lists the absolute timings for the original C++ benchmark, and the runtimes of the MCP versions relative to the original benchmark. The columns show respectively: 1) the name of the benchmark, 2) the parameter value (if any), 3) the runtime (in seconds) for the Gecode benchmark in C++, and the runtimes of 4) the C++ code generated in off-line mode, 5) the parametrized C++ code in off-line mode, 5) the on-line Gecode solver in fixed search mode, 6) the on-line Gecode solver in fixed programmed mode. All of columns 3)-6) are based on the same MCP model. The − entries denote a time out (no result after 80 seconds). Columns 7–9 show the corresponding compilation times and columns 9–12 show the number of lines of code.

Any timings close to the C++ runtime (column 3) indicate that the corresponding MCP mode is a valid alternative for direct Gecode implementation in terms of efficiency. The results for the naive Haskell solver are not shown here, but are at least an order of magnitude worse. Hence, it has been worthwhile to invest in Gecode backends for MCP.

A few times we observe that the compiled code generated in MCP off-line mode is slightly faster than the original Gecode benchmark. This is likely due to start-up overhead where the absolute runtime is only a few milliseconds.

These results show that more concise Haskell MCP programs can be used to generate C++ code whose performance matches that of original C++ implementations. When solving directly, there is an often small performance penalty when using the fixed search, and larger one when using programmed search.

5.2 Code Generation Results

Table 3 compares the generated code size and corresponding compilation times with those of an earlier version of FD-MCP, which did not yet support parametrization or higher-order constructs.

The results clearly support two conclusions:

1. Models written in MCP are more concise than in Gecode, and
2. Parametrized generated code avoids parameter-dependent code sizes.

6 Related and Future Work

There is wide range of CP systems and languages. We will mention a few. We subdivide them in stand-alone languages and API's. A more extensive overview of related work can be found in [10].

Stand-alone modeling languages Zinc [8] is a stand-alone modeling language. Model transformations and compilation processes to different constraint solver backends are implemented in a second language, Cadmium, which is based on ACD term rewriting [3].

Rules2CP [4] is another stand-alone modeling language. The compilation of Rules2CP to SICStus Prolog is also specified by rewrite rules.

name	size	Runtime (s)				Compile time (s)			Lines of code		
		C++	MCP			GCC		GHC	C++		Haskell
			gen.	search	run	C++	gen.	Haskell	orig.	gen.	
allinterval	7	0.004	0.004	0.0079	0.0096	1.3	0.87	0.035	52	89	22
	8	0.0045	0.0046	0.0091	0.014						
	9	0.0065	0.0069	0.012	0.031						
	10	0.016	0.018	0.024	0.11						
	11	0.066	0.077	0.08	0.5						
	12	0.32	0.38	0.37	2.5						
	13	1.8	2.1	2	13						
	14	10	12	12	-						
	15	61	-	-	-						
alpha			0.0045	0.018	0.023		1.1	0.047	0	257	39
bibd		0.004	0.0052	0.095	0.1	1.3	0.88	0.044	119	121	20
efpa		0.0044	0.005	0.065	0.067	1.8	0.96	0.045	199	154	19
golombruler	6	0.0041	0.0041	0.024	0.026	1.3	0.88	0.045	87	113	27
	7	0.0073	0.0081	0.035	0.056						
	8	0.042	0.047	0.08	0.23						
	9	0.35	0.4	0.41	1.6						
	10	3.2	3.5	3.2	12						
grocery		0.099	0.099	0.093	0.098	1.3	0.84	.032	42	77	12
magicseries	10	0.0039	0.0039	0.0093	0.0097	1.3	0.87	0.034	62	98	15
	20	0.004	0.004	0.015	0.016						
	50	0.0046	0.0043	0.032	0.036						
	75	0.0054	0.0048	0.048	0.054						
	100	0.0067	0.0053	0.063	0.074						
	200	0.016	0.0095	0.13	0.17						
	375	0.06	0.027	0.28	0.4						
	500	0.12	0.048	0.41	0.62						
	1000	0.94	0.36	1.4	2.2						

Table 1. Benchmarks

name	size	Runtime (s)				Compile time (s)			Lines of code		
		C++	MCP			GCC		GHC	C++		Haskell
			gen.	search	run	C++	gen.	Haskell	orig.	gen.	
magicsquare	3	0.0039	0.0039	0.011	0.011	1.3	0.87	0.044	62	95	27
	4	0.0081	0.0091	0.02	0.07						
	5	0.4	0.48	0.61	-						
	6	0.0042	0.0043	0.021	-						
	7	2.6	2.9	4.4	-						
partition	10	0.0044	0.0043	0.0096	0.02	1.4	0.9	0.046	74	105	28
	14	0.0086	0.0079	0.017	0.35						
	18	0.019	0.016	0.033	8.9						
	20	0.053	0.045	0.082	30						
	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	4.3	7	-						
	32	6.5	5.6	9.3	-						
queens	8	0.004	0.004	0.016	0.017	1.3	0.86	0.035	80	81	14
	10	0.004	0.0039	0.022	0.023						
	13	0.0043	0.0042	0.034	0.035						
	21	0.0042	0.004	0.08	0.13						
	34	0.0047	0.0044	0.2	-						
	55	0.0079	0.0075	0.52	-						
	70	0.0082	0.0073	0.84	-						
	89	0.012	0.011	1.4	-						
	100	0.015	0.013	1.7	-						
	111	0.089	0.089	2.2	-						

Table 2. Benchmarks (continued)

Benchmark		Lines of code			Compilation time(s)		
		C++	MCP		C++	MCP	
			param. C++	non-param. C++		param. C++	non-param. C++
allinterval	3	52	89	71	3.3	2.7	2.0
	15			179			2.7
queens	4	80	81	79	3.4	2.5	2.1
	100			534			>20
partition	4	74	105	103	3.3	3.1	2.2
	20			295			4.5
magicsquare	3	62	95	92	3.4	2.9	2.8
	6			206			3.1

Table 3. Lines of code and compilation times

Constraint Programming API's Two other functional programming languages which provide CP support are Alice ML[7] and FaCiLe [1]. While Alice ML provides a run-time interface to Gecode [6], FaCiLe uses its own constraint solver in OCaml. Both provide a rather low-level and imperative API, which corresponds to the C++ API of Gecode in the case of Alice ML, and relies on side effects. Neither supports alternative backends or model transformations.

Integrations Cipriano et al. [2] translate constraint models written in both Prolog CLP(FD) and (Mini)Zinc to Gecode via an intermediate language called CNT without loop constructs. The transformation from CNT to Gecode is implemented in Haskell. In order to avoid the Gecode code blow-up, it attempts to identify loops in the unrolled CNT model. It also performs a number of simplifications in the model. Our approach is much more convenient and efficient, providing explicit looping constructs and compiling these directly without intermediate loop unrolling, and with strong guarantees that loops remain loops.

Future work The support for lists should also be further extended to *multi-dimensional indexing*, which is quite convenient for modelling grid-based problems like sudoku, and *collection parameters* for providing a variable number of deferred data such as supply and demand quantities in a transportation problem.

One of the greatest challenges ahead of the MCP framework is the combination of programmed search and off-line solving. This combines the high-level search modeling with the performance advantages of C++ code by extending the staged compilation technique to include the search specification.

7 Conclusions

We have shown how to link the FD-MCP framework with Gecode to allow efficient on-line solving of constraint problems modeled using it. Furthermore, we added deferred parameters, indexable lists and higher-order list processing constructs to the provided abstractions, allowing shorter and more useful off-line code to be generated. These extensions were implemented[8] and benchmarks show that there is often only a small performance penalty compared to native C++ implementations.

Acknowledgments We are grateful to Peter Stuckey for his helpful comments.

References

1. N. Barnier. *Application de la programmation par contraintes à des problèmes de gestion du trafic aérien.* PhD thesis, Institut National Polytechnique de Toulouse, December 2002. `http://www.recherche.enac.fr/opti/papers/thesis/`.
2. R. Cipriano, A. Dovier, and J. Mauro. Compiling and executing declarative modeling languages to Gecode. In M. G. de la Banda and E. Pontelli, editors, *ICLP*, volume 5366 of *LNCS*, pages 744–748, 2008.

[7] `http://www.ps.uni-sb.de/alice`
[8] Available at `http://www.cs.kuleuven.be/~toms/MCP/`

3. 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.
4. F. Fages and J. Martin. From Rules to Constraint Programs with the Rules2CP Modelling Language. In *Recent Advances in Constraints*, LNAI, 2009.
5. A. J. Fernandez, T. Hortala-Gonzalez, F. Saenz-Perez, and R. Del Vado-Virseda. Constraint functional logic programming over finite domains. *Theory Pract. Log. Program.*, 7(5):537–582, 2007.
6. Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.
7. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.
8. K. Marriott et al. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.
9. S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.
10. T. Schrijvers, P. Stuckey, and P. Wadler. Monadic Constraint Programming. *J. Func. Prog.*, 19(6):663–697, 2009.
11. P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52, London, UK, 1995.
12. 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.
13. P. Wuille and T. Schrijvers. Expressive Models for Monadic Constraint Programming. In *Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation*, 2010. accepted.