The FD-MCP Framework

Pieter Wuille *Tom Schrijvers*

Report CW 562, August 2009



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

The FD-MCP Framework

Pieter Wuille *Tom Schrijvers*

Report CW 562, August 2009

Department of Computer Science, K.U.Leuven

Abstract

This report presents FD-MCP, a finite domain modeling language on top of the Monadic Constraint Programming framework for Haskell. FD-MCP leverages Haskell's rich static type system and powerful abstraction mechanisms for implementing syntactic sugar, model transformations and compilation to solver backends. Two backends are provided: a basic Haskell solver and a Gecode code generator.

Keywords : constraints, Haskell, MCP.
CR Subject Classification : D33

The FD-MCP framework

Pieter Wuille and Tom Schrijvers*

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

Abstract. This report presents FD-MCP, a finite domain modeling language on top of the Monadic Constraint Programming framework for Haskell. FD-MCP leverages Haskell's rich static type system and powerful abstraction mechanisms for implementing syntactic sugar, model transformations and compilation to solver backends. Two backends are provided: a basic Haskell solver and a Gecode code generator. Our benchmarks establish that FD-MCP models are much more concise than hand-coded Gecode programs, more so when using disjunction, without sacrificing performance.

1 Introduction

Modeling languages provide a high-level and declarative interface to one or more constraint solvers. Gecode [4] is a particularly attractive target for modeling languages: it is an efficient, open, free and portable constraint solving library for C++. At the same time, the imperative and low-level nature of C++ makes Gecode more suitable as a backend for a modeling language than a direct implementation language. It provides a predefined set of variable types, constraints, propagators and search strategies. Furthermore, it allows users to implement their own, without overhead compared to the built-in ones.

Because designing and developing a new (modeling) language from scratch is a significant effort, existing modeling languages often sacrifice established programming language features such as static typing or powerful abstraction mechanisms.

In this paper, we take a different approach and start from Haskell, a lazy purely functional language with state-of-the-art static typing and abstraction mechanisms [6]. Our finite domain (FD) modeling language, FD-MCP, is implemented as an embedded domain-specific language (EDSL) in Haskell. It extends the general Monadic Constraint Programming (MCP) [8] framework with FD-specific infrastructure.

The current state of our work in progress consists of:

- the FD-MCP modeling language embedded in Haskell,
- two backends for FD-MCP: a basic Haskell solver and the Gecode C++ library, and

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

– a compilation scheme to compile disjunctive FD models to Gecode, which provides no high-level support for disjunction,
– benchmarks that illustrate the succinctness of FD-MCP wrt. Gecode without sacrificing performance.

The FD-MCP code is available from `http://www.cs.kuleuven.be/~pieterw/site/Topics/FDMCP`.

2 Preliminaries

2.1 Haskell

Type Classes *Type classes* [10] are a Haskell feature heavily used in MCP. They allow defining classes of types using the `class` keyword. Class definitions state which functions and/or operators are needed for a type that implements that class. Contrary to OO programming languages, classes do not correspond to types themselves. For example:

```
class Eq a where
(==) :: a -> a -> Bool
```

This means: a type `a` belongs to the type class `Eq a`, if it provides the operator `==`, taking two arguments of type `a`, and returning a `Bool`. Providing an `instance` definition makes a type an instance of a class:

```
instance Eq ... where
  a == b = ...
```

Afterwards the `==` operator refers to the specific instance implementation, and works on any pair of arguments, provided they are both of a same type that belongs to the `Eq class`. For example:

```
allEqual :: Eq a => [a] -> Bool
allEqual [v] = True
allEqual a:b:r = (a==b) && allEqual (b:r)
```

The above function `allEqual` takes a list of any type `a`, on the condition that its elements belong to the type class `Eq` (as stated by `Eq a =>`).

Monads *Monads* [9] are an abstraction used in functional programming languages to represent computations. A monad `m` is a type parametrised in the type of the result of the computations, `a`. Instances of `m a` are called monad actions. Monads are defined by the type class:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

The function `return` must create a computation (or monad action) that simply represents a single given value. The `>>=` operator combines two Monad actions. It takes a monad action (producing something of type `a`), and a function turning something of type `a`, into a monad action returning something of type `b`, and results in a single monad action producing something of type `b`. The arrow `>>=` corresponds to the data flow: the left operand produces data, the right one consumes it.

In a pure, lazy functional language like Haskell, monads are amongst other things used for IO, which requires guarantees on the ordering of operations. For example, the functions `getLine` and `putStr` run in the IO monad, as they are of type `IO String` and `String -> IO ()`. `getLine` is a monad action that produces a `String` while `putStr` is a function that takes a `String`, and returns a monad action which returns nothing (type `()`). Combining them as `getLine >>= putStr` would result in a monad action of type `IO ()`, which represents reading a line of input, printing it out, and returning nothing.

Haskell provides syntactic sugar for composing monads. The following expressions left and right are equal. The first one reads a line, and prints it out again. The second one reads a line and ignores it, outputting "done". The `a >> b` notation is a shorthand for `a >>= (\_ -> b)`, equivalent to ignoring the output from the first action.

```
do x <- getLine              getLine >>= (\x ->
   putLine x                    putLine x)

do getLine                   getLine >> (
   putLine "done\n"            putLine "done\n")
```

`(\x -> putLine x)` is a lambda abstraction or anonymous function, taking a parameter `x`, and calling `putLine x`. In this case it is equivalent to just `putLine`, but works for arbitrarily complex expressions.

2.2 Monadic Constraint Programming

The MCP 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.

MCP defines type classes for `Solver`s and `Term`s:

```
class Monad s => Solver s where
  type Constraint s :: *
  add :: Constraint s -> s Bool
  run :: s a -> a
  ...

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

A type that implements the `Solver` type class must provide a type to represent its constraints, an `add` function for adding constraints, a `run` function to extract the results, ... A solver type s must also be a monad [9]. 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
 | Try (Model s a) (Model s a)            -- disjunction
 | Term s t => NewVar (t -> Model s a) -- introduce new variable
 | Dynamic (s (Model s a))                -- generate subtree
```

The model tree is parametrized in the constraint solver `s` and returned result `a`. This provides a type-safe way for representing constraint problem models for arbitrary solvers and result types. 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.

3 The FD-MCP Framework

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 different 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.

```
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   a @>= b
6   b @>= c
7   c @>= d
```

Fig. 1. Grocery example in MCP

3.1 The Modeling Frontend

Figure 1 illustrates the 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. [1]

Line 1 starts the declaration of an FD-MCP model tree `model`. `exist` requests four new variables, and has an (anonymous) callback function for handling those new variables as parameter. This function itself takes the `list` of new variables as argument, individually aliased as `a`, `b`, `c` and `d`. Lines 2–7 define a conjunction of constraints that must hold for these variables:

- Line 2: restrict all variables' domains to $0 \ldots 711$.
- Line 3: the sum of the four variables is 711.
- Line 4: the product of the four variables is 711000000.
- Line 5–7: each variable is larger than or equal to the next one (to break symmetry).

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 `exist` 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 small set of core primitives, resulting in a layered structure.

The core constraints and terms are defined by the `FDConstraint` and `FDExpr` types respectively:

```
data FDConstraint s
 = Less (FDExpr s) (FDExpr s)   -- [[Less x y]]   ≡ [[x]] < [[y]]
 | Diff (FDExpr s) (FDExpr s)   -- [[Diff x y]]   ≡ [[x]] ≠ [[y]]
 | Same (FDExpr s) (FDExpr s)   -- [[Same x y]]   ≡ [[x]] = [[y]]
 | Dom (FDExpr s) Int Int       -- [[Dom x y z]]  ≡ [[x]] ∈ {y,...,z}
 | AllDiff [FDExpr s]           -- [[AllDiff [x1,...,xn]]] ≡ ⋀_{i≠j} xi ≠ xj
```

[1] 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.

```
 | ...

data FDExpr s
 = Var (FDTerm s)                -- ⟦Var v⟧       ≡ ⟦v⟧
 | Const Int                     -- ⟦Const n⟧     ≡ n
 | Plus (FDExpr s) (FDExpr s)    -- ⟦Plus x y⟧    ≡ ⟦x⟧ + ⟦y⟧
 | Minus (FDExpr s) (FDExpr s)   -- ⟦Minus x y⟧   ≡ ⟦x⟧ - ⟦y⟧
 | Mult (FDExpr s) (FDExpr s)    -- ⟦Mult x y⟧    ≡ ⟦x⟧ * ⟦y⟧
 | ...
```

where the comment after each constructor shows its denotation.

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 `FDExpr s` thanks to an implementation of Haskell's `Num` type class:

```
instance Num (FDExpr s) where
  fromInteger = Const
  (+)         = Plus
  ...
```

Thus `Plus x (Mult (Const 2) y)` can be written succinctly as `x + 2 * y`. More syntactic sugar exists for writing constraints:[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 ]
...
```

The constraint operators are in fact a little smarter than that. Namely, the first one is defined as:

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

simplify (Plus x (Const 0)) = simplify x
simplify ...             = ...
```

i.e. `@<` acts as a smart constructor that first simplifies its arguments. Argument simplification is a solver-independent model transformation built into our framework. It heuristically minimizes the number of variable occurrences and the number of nodes in the expression tree. Of course, programmers can define their own libraries of abstractions and model transformations in entirely the same style.

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

Example 1. Figure 2 lists the model obtained from Fig. 1 after desugaring the abstractions. The model consists of a first `NewVar` node, which has an anony-

```
NewVar $ \a ->
  NewVar $ \b ->
    NewVar $ \c ->
      NewVar $ \d ->
        Add (Dom a 0 711) $
          Add (Dom b 0 711) $
            Add (Dom c 0 711) $
              Add (Dom d 0 711) $
                Add (Same (Plus a (Plus b (Plus c d))) (Const 711)) $
                  Add (Same (Mult a (Mult b (Mult c d))) (Const 711000000)) $
                    Return ()
```

Fig. 2. FD model tree of the grocery example (see Figure 1)

mous function as parameter (after the $). When passed a variable (of type `FDTerm s`, assuming the underlying solver is `s`), this function evaluates to a new `NewVar` node, again having a function as parameter. This pattern repeats itself four times, corresponding to the four variables introduced. The parameter of the fourth-level `NewVar` evaluates to a tree of `Add` nodes, each having an `FDConstraint` as one parameter, and the rest of the tree as child node. First four `Dom` constraints are added (corresponding to the `allin` in the model source code), followed by two `Same` constraints (corresponding to the two `@=` constraints), and finally three `Less` constraints. The `Return` node returns unit `()`, i.e., no value of interest.

3.2 Mapping to the solver backend

The backend takes care of compiling an FD-MCP model to a particular FD solver. To enable this compilation the solver must implement the following `FDSolver` type class (in addition to implementing the `Solver` type class of the MCP framework):

```
class Term s (FDTerm s) => FDSolver s where
  type FDTerm s :: *
  compile_constraint :: FDConstraint s -> Model s Bool
```

The `FDSolver` type class makes two demands of a solver `s`:

- It must provide a type `FDTerm s` for its terms.
- The function `compile_constraint` must take care of converting from an individual FD-MCP constraint to a model for the solver. Note that, to allow mapping a single FD-MCP constraint to a conjunction of solver constraints involving auxiliary variables, this function returns a model rather than a single constraint. This model is not allowed to contain any disjunctions.

The following law specifies the `compile_constraint` function:

Definition 1 (Denotation Preservation). *The `compile_constraint` function preserves denotation iff*

$$[\![\cdot]\!] \circ \mathit{compile_constraint} \equiv [\![\cdot]\!]$$

where $[\![\cdot]\!]$ maps a model or constraint onto its denotation, i.e. its logical meaning, and $\equiv$ denotes extensional function equality. Two denotations are equal iff they are logically equivalent.

The wrapper's constraints `FDConstraint s` are those of the generic model, not those of the underlying solver `s`. When adding such a constraint, it is first compiled to a model for the underlying solver and then this model is added using the auxiliary `untree` function, which turns the model into a solver monad.

3.3 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 a Solver in the MCP framework which uses `FDConstraint`s as constraints and `FDExpr`s as terms.

```
1  newtype FDWrapper s a = FDWrapper { unwrapFD :: s a }
2
3  instance FDSolver s => Solver (FDWrapper s) where
4    type Constraint (FDWrapper s) = FDConstraint s
5    add c = FDWrapper $ untree $ compile_constraint c
6    ...
7    run = run . unwrapFD
8
9  instance Term (FDWrapper s) (FDTerm t) where
10   newvar = FDWrapper $ newvar >>= \x -> return $ Var x
11   ...
12
13 untree :: Solver s => Model s a -> s a
14 untree ... = ...
```

The `add` function first converts one `FDConstraint s` to a model tree for the underlying solver `s` with the `compile_constraint` function. Then, it turns this tree into a single (wrapped) monad action for the underlying solver `s`. Not doing this at once inside `compile_constraint` as the advantage of allowing deeper inspection of the result. It is in fact required for the `gecode_compile` function (see Section 5.3).

A similar approach is used for the other solver methods. The `newvar` method of the wrapper requests a new variable from the underlying solver, and returns it wrapped in a `Var` constructor. This is why the `FDConstraint` and `FDExpr` structures, as well as `FDWrapper` itself, are parametrized in the underlying MCP Solver `s`. Finally, `run` unwraps the encapsulated monad action and runs it.

4 The Overton Instance

The Overton FD solver is a basic finite domain solver in Haskell. It has been adapted from an implementation by David Overton[3] to instantiate the FD-MCP framework.

Firstly, a `Solver` instance incorporates the Overton FD solver in the general MCP framework:

```
instance Solver OvertonFD where
  type Constraint OvertonFD  = OConstraint
  add c          = addFD c
  run p          = runFD p
  ...

instance Term OvertonFD FDVar where ...
```

where `FDVar` is the type of terms and `OConstraint` represents the supported constraints:

```
data OConstraint
 = OHasValue FDVar Int       -- [[OHasValue v c]] ≡ [[v]] = c
 | OSame FDVar FDVar         -- [[OSame x y]] ≡ [[x]] = [[y]]
 | ODiff FDVar FDVar         -- [[ODiff x y]] ≡ [[x]] ≠ [[y]]
 | OLess FDVar FDVar         -- [[OLess x y]] ≡ [[x]] < [[y]]
 | OAdd FDVar FDVar FDVar    -- [[OAdd x y z]] ≡ [[x]] + [[y]] = [[z]]
 | OSub FDVar FDVar FDVar    -- [[OSub x y z]] ≡ [[x]] - [[y]] = [[z]]
 | OMult FDVar FDVar FDVar   -- [[OMult x y z]] ≡ [[x]] * [[y]] = [[z]]
```

The `FDSolver` instance also enables the FD modeling language:

```
instance FDSolver OvertonFD where
  type FDTerm OvertonFD = FDVar
  compile_constraint = convert
```

Note that the `OvertonFD` solver's term language is much more restricted than that of the generic FD-MCP modeling language: it only provides single variables (`FDVar`), compared to the more extensive `FDExpr`-expressions. The translation `convert` overcomes this syntactic mismatch.

```
convert :: FDConstraint OvertonFD -> Model OvertonFD ()
convert (Less a b) = do va <- decomp a
                        vb <- decomp b
                        addC $ OLess va vb
convert ...

decomp :: FDExpr OvertonFD -> Model OvertonFD FDVar
decomp (Var f)  = return f
decomp (Const i) = exists $ \v -> addC (OHasValue v i) /\ return v
```

[3] `http://overtond.blogspot.com/2008/07/pre.html`

```
decomp (Plus a b) = do
  va <- decomp a
  vb <- decomp b
  exists $ \v -> addC (OAdd va vb v) /\ return v
decomp ...
```

The auxiliary function `decomp` decomposes a compound `FDExpr` into a model tree of `OConstraint`s over `FDVar`s, by using the `addC` helper function to add constraints to the solver. At the same time it introduces a fresh constraint variable for each subexpression. This allows using the compact modeling language of the FD framework, although the `OvertonFD` solver only provides a much more primitive language.

Note that `compile_constraint` conforms to the law given in Definition 1. For example, assume an input expression `Plus a b`. Its denotation is $[\![\texttt{a}]\!] + [\![\texttt{b}]\!]$. Running `decomp` on it, results in an output expression `v` combined with the additional constraint `OAdd va vb v`, where `va` and `vb` are the result of applying `decomp` on `a` and `b` recursively. Assuming by induction that $[\![\texttt{va}]\!] \equiv [\![\texttt{a}]\!]$ and $[\![\texttt{vb}]\!] \equiv [\![\texttt{b}]\!]$, $[\![\texttt{OAdd va vb v}]\!] \equiv [\![\texttt{v}]\!] = [\![\texttt{va}]\!] + [\![\texttt{vb}]\!]$ leads to $[\![\texttt{v}]\!] \equiv [\![\texttt{a}]\!] + [\![\texttt{b}]\!]$.

Example 2. The translation of the the sum and product constraints from the FD model given in Figure 2, is shown in Figure 3. Both constraints are translated to a combination of `NewVar` and `Add` nodes. All subexpressions in the original FD model have been decomposed into an introduction of a new variable and an additional constraint. The variables `a`, `b`, `c`, `d` refer to the original constraint variables, while `i1`, `i2`, `i3` are auxiliary variables, introduced by the translation.

5 The Gecode Instance

Rather than to run an FD-MCP model on a solver in Haskell, this section shows how to run it on a solver implemented in C++, namely Gecode. This approach illustrates three practical advantages of FD-MCP. Firstly, the framework is able to reuse existing FD solvers implemented in other programming languages. Secondly, this approach benefits from the performance characteristics of a C++ implementation. Thirdly, the high-level modeling language of the framework becomes available for low-level solvers.

The approach consists of three layers:

1. The FD-MCP solver-independent modeling language embedded in Haskell, presented earlier in Section 3.
2. The `Gecode` solver abstraction in Haskell.
3. The Gecode library in C++.

This section contributes the second layer, as well as the mappings from the first layer and to the third layer.

5.1 The Intermediate Layer: the `Gecode` Solver

The `Gecode` solver is a Haskell abstraction of the actual Gecode C++ solver. It defines the syntax of constraints and terms understood by Gecode:

```
instance Term Gecode IntTerm where ...

instance Solver Gecode where
  type Constraint Gecode = GConstraint
  ...
```

Gecode terms are either integer variables or integer constants:

```
data IntTerm
 = IntVar Int     -- reference to variable with given ID
 | IntConst Int   -- constant integer
```

The language of constraints is richer than that of the Haskell constraint solver:

```
data GConstraint
 = CRel IntTerm GOper IntTerm     -- [[CRel x op y]] ≡ [[x]] [[op]] [[y]]
 | CDom IntTerm Int Int           -- [[CDom x y z]] ≡ [[x]] ∈ {y,...,z}
 | CMult IntTerm IntTerm IntTerm  -- [[CMul x y z]] ≡ [[x]] * [[y]] = [[z]]
 | CAllDiff [IntTerm]
                     -- [[CAllDiff [x1,...,xn]]] ≡ ⋀_{i≠j} x_i ≠ x_j
 | CLinear [(IntTerm,Int)] GOper Int
                     -- [[CLinear [(x1,a1),...] op b]] ≡ (Σ_i a_i * [[x_i]]) [[op]] c

data GOper
 = GEqual     -- [[GEqual]] ≡ =
 | GDiff      -- [[GDiff]]  ≡ ≠
 | GLess      -- [[GLess]]  ≡ <
```

Note in particular the support for linear constraints `CLinear` and the all-different global constraint `CAllDiff`.

5.2 From Generic FD Model to Gecode-Specific Model

In order to convert generic FD models to Gecode specific ones, the `Gecode` solver implements the `FDSolver` type class:

```
instance FDSolver Gecode where
  type FDTerm Gecode = IntTerm
  compile_constraint = mixin (linear_compile <@> basic_compile)
```

The compilation of constraints mixes two strategies. First, `linear_compile` attempts to map the `FDConstraint` onto a single linear `Gecode` constraint. If this fails, `basic_compile` decomposes the constraint, and the overall strategy is applied recursively to the decomposed parts.

```
Same (Plus a (Plus b (Plus c d))) (Const 711)
```

```
NewVar $ \i1 ->
  Add (CAdd c d i1) $
    NewVar $ \i2 ->
      Add (CAdd b i1 i2) $
        NewVar $ \i3 ->
          Add (CAdd a i2 i3) $
            Add (CHasValue i3 711) $
              Return ()
```

```
Same (Mult a (Mult b (Mult c d))) (Const
                                   711000000)
```

```
NewVar $ \i1 ->
  Add (CMult c d i1) $
    NewVar $ \i2 ->
      Add (CMult b i1 i2) $
        NewVar $ \i3 ->
          Add (CMult a i2 i3) $
            Add (CHasValue i3 711000000) $
              Return ()
```

```
Add (CLinear
    [(a,1),(b,1),(c,1),(d,1)]
    GEqual
    711
  ) $ Return ()
```

```
NewVar $ \i1 ->
  Add (CMult c d i1) $
    NewVar $ \i2 ->
      Add (CMult b i1 i2) $
        NewVar $ \i3 ->
          Add (CMult a i2 i3) $
            Add (CRel i3 GEqual 711000000) $
              Return ()
```

Fig. 3. Translation of the sum (left) and product (right) constraints in the grocery example for the built-in Haskell solver (top) and the Gecode solver (bottom).

Example 3. Figure 3 shows the translation of the sum and product constraints from the grocery model. The result for the product constraint is comparable to that for the Haskell solver. The sum constraint however, is converted to a single `CLinear` constraint, without any decomposition.

5.3 From Haskell to C++

The final step consists of mapping the `Gecode` solver abstraction in Haskell to the actual Gecode library in C++. It follows a code generation approach. The `gecode_compile :: Model Gecode a -> String` function transforms a `Gecode` model into the equivalent C++ source code that calls the appropriate Gecode library functions. For instance, Figure 6 in Appendix A lists the generated C++ code for the 7-11 model of Figure 1.

5.4 Disjunctive Models

Compilation of disjunctive models to Gecode is not straightforward. While general disjunction is a primitive in the MCP modeling language, it is not in Gecode.

As a work-around, we have first considered compiling constraints c in disjunctive branches to reified constraints $c \Leftrightarrow b$ in Gecode. A reified constraint One way to support them would be to turn all constaints posted belonging to one of the disjunctive branches of the tree into reified constraints. This means a variant of simple constraints that take an additional boolean variable as argument,

whose value shall be true if the original constraint holds, and false otherwise. Unfortunately, Gecode does not have reified versions of all supported constraints.

Currently only a default compilation scheme is implemented which relies on Gecode's `when` co-routine: `when(`b`,`f_t`,`f_f`)` causes an invocation of the C++ function f_t when the variable b becomes fixed to *true*, and of f_f when it becomes *false*. In the future, particular cases will be compiled to the more efficient reified constraints.

Figure 4 shows the translation of this model:

```
demo = exists $ \a -> a @= 1 \/ a @= 2
```

or alternatively, its resulting model tree:

```
demo = NewVar $ \a ->
  Try
    (Add (Same a (Const 1)) $
      Return ())
    (Add (Same a (Const 2)) $
      Return ())
```

```
1   ...
2   BoolVarArray i;
3   ...
4   HaskellProg() : v0(*this,1,2), i(*this,1,0,1) {
5     node(*this);
6     branch(home, p->i, INT_VAR_SIZE_MIN, INT_VAL_MIN);
7     IntVarArgs bIntVar(1);
8     bIntVar[0]=p->v0;
9     branch(home, bIntVar, INT_VAR_SIZE_MIN, INT_VAL_MIN);
10  }
11  static void node(Space &home) {
12    HaskellProg *p = (HaskellProg *)(&home);
13    when(home,p->i[0],&nodeR,&nodeL);
14  }
15  static void nodeL(Space &home) {
16    HaskellProg *p = (HaskellProg *)(&home);
17    rel(home,p->v0,IRT_EQ,1);
18  }
19  static void nodeR(Space &home) {
20    HaskellProg *p = (HaskellProg *)(&home);
21    rel(home,p->v0,IRT_EQ,2);
22  }
23  ...
```

Fig. 4. Gecode C++ code generated for the disjunctive demo model

Line 2 introduces an additional branch-selection array of booleans `i`, here with size 1. Lines 4–10 show the new constructor. Compared to the compilation scheme used in Figure 6, it posts an additional branching over `i` prior to posting the real branching. The actual constraints are moved from the constructor to the

(root) node function `node`, shown in lines 11–14. Worth noticing is line 13 which posts the `when` constraint, passing references to functions `nodeL` and `nodeR`. The rest of the code (lines 15–22) shows the subnode functions. Both only post a single real constraint in this case.

More generally, the compilation introduces an array of boolean variables, one for each level in the disjunctive tree. For every branch and leaf a specific function is generated in the output code. Functions for non-leaf nodes post a `when` constraint, passing pointers to the functions corresponding to its two child branches. Functions for leaf-nodes above the lowest tree depth post a dummy constraint over the remaining branch-selection variables, to prevent useless search over them. These functions contain all postings of constraints. The main function first calls the function for the root node, and proceeds by posting a branching over the branch-selection array and and one over the model variables. That way, search first iterates over all branches of the disjunctive tree, storing and restoring the part of the internal state that is common to multiple branches.

5.5 Implicit Constraint Variables

As stated in Section 3.2, and shown by previous examples, the `compile_constraint` function may generate a complete model tree (without disjunctions) for a single constraint. Such a tree may introduce auxiliary constraint variables, not part of the original model. These automatically introduced, or *implicit*, constraint variables raise three concerns:

1. The programmer is not interested in these variables since they don't exist in the original model, and would like to have them excluded from the produced results.
2. Many FD solvers—including Gecode—expect an initial domain for their constraint variables. In the absence of an initial domain, propagators may diverge or take altogether too much time. The implicit variables pose a problem because the programmer has obviously not provided them with an initial domain.
3. Gecode frees variables that are no longer needed. Knowing which variables are not part of a solution allows generating code that does not keep them longer than necessary.

Identifying Implicit Variables To distinguish these implicit variables from those added explicit by `exists`, the model tree needs support for tagging them. An elegant way for this is using the `Dynamic` node. Such a node contains a monad action for the solver, called at run time, producing the subtree dynamically — potentially depending on the actual solver state.

The helper function `auxiliary` which produces a new implicit variable. It uses `registerImplicit` which tags a variable as implicit.

```
auxiliary :: Model Gecode IntTerm
auxiliary = exists $ \v -> Dynamic $ do
```

```
  registerImplicit v
  return $ Return v
```

Each time `compile_constraint` needs a new variable, it uses `auxiliary` instead of `exists`. This way, the resulting state knows which variables are implicit and which aren't. For instance, `Plus` is translated as follows:

```
getAsVar (Plus a b) = do
  n <- auxiliary
  v1 <- getAsVar a
  v2 <- getAsVar b
  addC (CLinear [(v1,1),(v2,1),(n,-1)] GEqual 0)
  return n
```

Thus an expression (`Plus a b`), is translated by first generating a new implicit variable, decomposing the subexpressions `a` and `b` into variables (possibly adding constraints to the solver tree in the process), adding a `CLinear` constraint stating $v_1 + v_2 - n = 0$, and finally returning the new implicit variable.

Initial Domain Inference It is the `Gecode` solver's responsibility to provide an initial domain for the implicit variables it introduces. Thus it statically infers the lower and upper bounds, using the posted constraints and the earlier bounds of the variables referenced therein.

The inference is comparable to dynamic constraint propagation, except for avoiding unbounded computations (e.g. variable ranges are represented as a lower and upper bound only). In contrast, the inference combines information across disjunctions in the search tree.

Finally, note that the inference is not restricted to the implicit variables. It also applies to the explicit constraint variables, and attempts to improve programmer-supplied initial domains.

6 Evaluation

Five existing benchmarks for Gecode were used and ported to FD-MCP. Table 1 shows the results. The columns show respectively the name of the benchmark, the problem size (if any), and the results for 1) the original Gecode benchmarks, 2) the Haskell part of the MCP benchmarks and 3) the C++ part of the MCP benchmarks. The results for each stage consist of the number of source code lines, the compilation time and the run time. Lines of code are measured using SLOCcount[4], the timings are average CPU times over multiple runs.[5]

The table shows that the MCP programs are about half as long as the original Gecode benchmarks, while the generated C++ code is much longer and grows with the problem size. However, this does not result in lower performance.

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

[5] Benchmarks are performed on a 64-bit Ubuntu 9.04 system using a 1.67GHz Intel® Core™2 Duo T5500 processor, with 1GiB RAM. Software versions: GHC 6.10.3, GNU G++ 4.3.3, Gecode 3.1.0.

name	size	Gecode C++			MCP Haskell			MCP C++		
		loc	g++	run	loc	ghc	run	loc	g++	run
allinterval	10	52	2.Gs	0.045s	25	1.0s	0.038s	138	2.4s	0.049s
	15			116s			0.063s	183	2.7s	129s
grocery	-	42	2.4s	0.23s	23	0.86s	0.007s	72	2.0s	0.23s
nqueens	4	80	2.5s	0.005s	25	1.0s	0.007s	79	2.1s	0.005s
	8			0.006s			0.023s	161	3.7s	0.006s
	12			0.006s			0.058s	291	6.7s	0.006s
	18			0.006s			0.27s	576	16s	0.006s
	24			0.007s			0.95s	969	42s	0.006s
magicsquare	4	62	2.6s	0.019s	34	1.0s	0.017s	122	2.4s	0.019s
	6			0.006s			0.055s	206	3.1s	0.006s
	7			4.8s			0.062s	260	3.7s	4.8s
partition	8	74	3.1s	0.007s	39	1.2s	0.027s	151	3.3s	0.006s
	16			0.045s			0.091s	247	4.3s	0.049s
	24			1.2s			0.23s	343	5.6s	1.4s

Table 1. Benchmarks

7 Related and Future Work

High-level modeling languages are an active topic of research. For lack of space, we can only mention a few systems.

Zinc [5] is a stand-alone modeling language; unlike FD-MCP's host language Haskell, Zinc is not Turing-complete. Model transformations and compilation processes to different constraint solver backends are implemented in a second language, Cadmium, which is based on ACD term rewriting [2]. In contrast, FD-MCP captures all these transformation steps in one single language, Haskell.

Extensions exist [7] that allow declarative specifcation of the search procedure. There are two simpler languages derived from it: MiniZinc and FlatZinc. The CLP system ECLiPSe [11] currently supports embedding MiniZinc models. uit G12 readme: MiniZinc models can be embedded into ECLiPSe code in order to add user-defined search and I/O facilities to the models.

Cipriano et al. [1] 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 on the model.

Rules2CP [3] is another stand-alone modeling language. Rules2CP is compiled to SICStus Prolog, and is not Turing-complete.

In future work, we would like to improve upon Cipriano et al.'s approach to directly map loop constructs from FD-MCP to Gecode, without the need for intermediate loop unrolling. A second major challenge is to generate efficient Gecode search strategies from high-level specifications using MCP's composable search transformer infrastructure.

We plan to implement a third instance of the FD-MCP framework that acts as a runtime frontend to the Gecode library, and directs the search from Haskell using its Foreign Function Interface, avoiding intermediate compilation steps.

8 Conclusions

Figure 5 summarizes FD-MCP, a new layer on top of the MCP framework. FD-MCP provides a high-level declarative FD modeling language with multiple backends. Two backends are implemented: 1) a pure Haskell FD solver, and 2) a Gecode code generator. The latter brings support for high-level disjunctive models to Gecode. Benchmarks show that FD-MCP models are significantly more compact than hand-written Gecode programs, without seriously affecting performance.

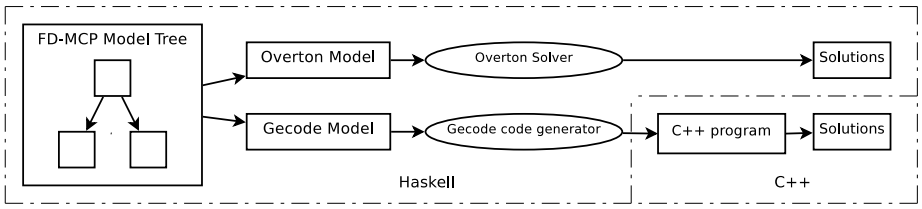

Fig. 5. FD-MCP schematic overview.

Acknowledgments We are grateful to Christian Schulte for his helpful comments.

References

1. 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.
2. 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.
3. F. Fages and J. Martin. From Rules to Constraint Programs with the Rules2CP Modelling Language. In *Recent Advances in Constraints*, LNAI, 2009.
4. Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.
5. K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. Garcia De La Banda, and M. Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.
6. S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.

7. R. Rafeh, K. Marriott, M. G. Banda, N. Nethercote, and M. Wallace. Adding search to Zinc. In *CP '08: Proceedings of the 14th international conference on Principles and Practice of Constraint Programming*, pages 624–629, 2008.
8. T. Schrijvers, P. Stuckey, and P. Wadler. Monadic constraint programming. *Journal of Functional Programming*, 2009.
9. P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995.
10. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM.
11. M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming, 1997.

A Generated C++ Example

```
#include ...

using namespace Gecode;

class HaskellProg : public Space {
protected:
  IntVar v0, v1, v2, v3, v4, v5, v6, v7;

public:
  HaskellProg() : v0(*this,1,711), v1(*this,1,711), v2(*this,1,711),
    v3(*this,1,711), v4(*this,711000000,711000000), v5(*this,1,359425431),
    v6(*this,1,505521), v7(*this,711000000,711000000) {
    node(*this);
    HaskellProg *p = (HaskellProg *)(this);
    { IntVarArgs iva(4); iva[0]=p->v1; iva[1]=p->v2; iva[2]=p->v3;
      iva[3]=p->v0;  linear(home,iva,IRT_EQ,711); };
    mult(home,p->v3,p->v2,p->v6);
    mult(home,p->v6,p->v1,p->v5);
    mult(home,p->v5,p->v0,p->v4);
    rel(home,p->v4,IRT_EQ,p->v7);
    { IntArgs ia(2,1,-1); IntVarArgs iva(2); iva[0]=p->v2; iva[1]=p->v3;
      linear(home,ia,iva,IRT_LE,1); };
    { IntArgs ia(2,1,-1); IntVarArgs iva(2); iva[0]=p->v1; iva[1]=p->v2;
      linear(home,ia,iva,IRT_LE,1); };
    { IntArgs ia(2,1,-1); IntVarArgs iva(2); iva[0]=p->v0; iva[1]=p->v1;
      linear(home,ia,iva,IRT_LE,1); };
    IntVarArgs bIntVar(4);
    bIntVar[0]=p->v0; bIntVar[1]=p->v1; bIntVar[2]=p->v2;
    bIntVar[3]=p->v3;
    branch(home, bIntVar, INT_VAR_SIZE_MIN, INT_VAL_MIN);
  };
  virtual void print(std::ostream& os) { ... }
  HaskellProg(bool share, HaskellProg &s) : Space(share,s) { ... }
  virtual Space* copy(bool share) { ...  }
};

int main(void) {
  HaskellProg *prog=new HaskellProg();
  DFS<HaskellProg> srch(prog);
  delete prog;
  do {
    HaskellProg *sol=srch.next();
    if (sol==NULL) break;
    sol->print(std::cout);
  } while(1);
  return 0;
}
```

Fig. 6. Gecode C++ code generated for the 7-11 model