Integration of Finite Domain Constraints in KiCS2 *

Michael Hanus Björn Peemöller Jan Rasmus Tikovsky

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{mh|bjp|jrt}@informatik.uni-kiel.de

Abstract: A constraint programming system usually consists of two main components: a modelling language used to describe a constraint satisfaction problem and a constraint solver searching for solutions to the given problem by applying specific algorithms. As constraint programming and functional logic languages share some common features, like computing with logic variables or the use of backtracking for non-deterministic search, it is reasonable to embed a modelling language for finite domain constraints in a functional logic language like Curry. Due to the absence of side effects or global state over non-deterministic computations in Curry, the implementation of a stateful constraint solver is rather difficult. In this paper we consider KiCS2, a Curry compiler translating Curry programs into Haskell programs. In order to embed finite domain constraints in KiCS2, we propose a new implementation technique compatible with the purely functional nature of its back end. Our implementation collects finite domain constraints occurring during a program run and passes them to a constraint solver available in Haskell whenever solutions are requested.

1 Introduction

Functional logic languages combine the most important features of functional and logic languages (more information can be found in recent surveys [AH10, Han13]). They support functional concepts like higher-order functions and lazy evaluation as well as logic programming concepts like non-deterministic search and computing with partial information. This combination allows better abstractions for application programming and has also led to new design patterns [AH11] and language extensions [AH05, AH09]. The objective of such developments is the support of a high-level and declarative programming style.

Another approach to solve complex problems in a declarative manner is constraint programming [MS98]. It is based on the idea to describe the problem to be solved by properties, conditions, and dependencies (i.e., *constraints*) between the parameters of the problem. Based on this declarative description, a constraint solver tries to find one or all solutions satisfying these constraints. Although the problem description is declarative,

the constraint solver usually works in a stateful manner by manipulating an internal constraint store. The constraint store is initially filled with the constraints specified by the programmer. The solver then searches for solutions by iteratively simplifying constraints and making guesses until it reaches either a solution or an unsatisfiable constraint. In the latter case, the solver will recover to a consistent state to explore the remaining alternatives (backtracking). A special but practically relevant case of constraint programming is finite domain (FD) constraint programming where variables range over a finite domain, like Boolean values or a finite subset of integer values.

Functional logic languages like Curry [He12] or TOY [LFSH99] allow a declarative style of programming and support logic variables and non-deterministic search. Thus, it is reasonable to embed FD constraints in these languages. Actually, this has already been proposed [FHGSPdVV07] and implemented in Curry [AH00] as well as in TOY [FHGSP03]. These implementations exploit FD constraint solvers in Prolog, i.e., they are limited to implementations with a Prolog-based back end.

In this paper, we are interested in the support of FD constraints in a non-Prolog-based implementation of Curry. In particular, we consider KiCS2 [BHPR11], a recent compiler translating Curry programs into purely functional Haskell programs using the Glasgow Haskell Compiler (GHC) as its back end. The motivation to use KiCS2 is the fact that it produces much more efficient target programs (see the benchmarks of Brassel et al. [BHPR11]) and supports more flexible search strategies [HPR12] than Prolog-based implementations of Curry.

Due to the absence of side effects and a global state over non-deterministic computations, it is not obvious how to implement an FD constraint solver in this environment. However, the monadic constraint programming (MCP) framework [SSW09] provides an implementation of a stateful FD constraint solver in Haskell using the concept of monads. In order to reuse this framework to offer FD constraints in KiCS2, we extend the compilation scheme of KiCS2 to collect FD constraints occurring during a (potentially non-deterministic) program run and solve them using the MCP framework as the solver back end.

The rest of the paper is structured as follows. We review the source language Curry in Sect. 2 and the features of FD constraints considered in this paper in Sect. 3. Sect. 4 introduces the implementation scheme of KiCS2, whereas Sect. 5 presents our implementation of an FD constraint library in KiCS2 and the integration of the solvers of the MCP framework. We evaluate our implementation in Sect. 6 before we conclude in Sect. 7.

2 Curry

The syntax of the functional logic language Curry [He12] is close to Haskell [PJ03], i.e., type variables and names of defined operations usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of an operation f to an expression e is denoted by juxtaposition ("$f\ e$"). In addition to Haskell, Curry allows free (logic) variables in conditions and right-hand sides of defining rules. Hence, an operation is generally defined by conditional rewrite rules of the form

```
f t1 ... tn | c = e where vs free
```

where the *condition* c is optional and vs is the list of variables occurring in c or e but not in the *left-hand side* $f\ t_1 \ldots t_n$. The condition c is a *constraint*, i.e., an expression of type `Success`. An elementary constraint is an *equational constraint* of the form e_1 `=:=` e_2 which is satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms. Furthermore, "c_1 `&` c_2" denotes the conjunction of the constraints c_1 and c_2. For instance, the following program defines a few operations on Boolean values and a relation between four inputs describing the functionality of a half adder circuit:

```
not True  = False          xor True  x = not x       and False x = False
not False = True           xor False x = x           and True  x = x

halfAdder x y sum carry  =  sum =:= xor x y  &  carry =:= and x y
```

With these definitions, the evaluation of the goal "`halfAdder x y sum True`" yields the single solution $\{$`x` $=$ `True`, `y` $=$ `True`, `sum` $=$ `False`$\}$.

In contrast to functional programming and similarly to logic programming, operations can be defined by overlapping rules so that they might yield more than one result on the same input. Such operations are also called *non-deterministic*. For instance, Curry offers a *choice* operation that is predefined by the following rules:

```
x ? _ = x
_ ? y = y
```

Thus, we can define a non-deterministic operation `aBool` by

```
aBool = True ? False
```

so that the expression `aBool` has two values: `True` and `False`. If non-deterministic operations are used as arguments in other operations, a semantical ambiguity might occur. Consider the operation

```
xorSelf x = xor x x
```

and the expression "`xorSelf aBool`". If we interpret the program as a term rewriting system, we could have the reduction

```
xorSelf aBool  →  xor aBool aBool  →  xor True  aBool
               →  xor True  False  →  not False       →  True
```

leading to the unintended result `True`. Note that this result cannot be obtained if we use a strict strategy where arguments are evaluated prior to the function calls. In order to avoid dependencies on the evaluation strategies and exclude such unintended results, the rewriting logic CRWL is proposed by González-Moreno et al. [GMHGLFRA99] as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and non-deterministic operations. This logic specifies the *call-time choice* semantics [Hus92], where values of the arguments of an operation are determined before the operation is evaluated. In a lazy strategy, this can be enforced by sharing actual arguments. For instance, the expression above can be lazily evaluated provided that all occurrences of `aBool` are shared so that all of them reduce to either `True` or `False` consistently.

3 Finite Domain Constraint Programming

Finite domain (FD) constraint programming aims to solve complex combinatorial problems in a high-level manner [MS98]. The problems to be solved are specified by properties and conditions (constraints). These constraints are typically defined by declarative rules and relations for variables that range over a finite domain of values. Due to its declarative nature, constraint programming can be naturally integrated into declarative programming languages, like logic programming [JL87] or functional logic programming [FHGSPdVV07] languages.

A constraint programming system usually consists of two main components: a modelling language used to describe a constraint satisfaction problem and a constraint solver searching for solutions to the given problem by applying specific algorithms. The solver has an internal state which is determined by the satisfiability of the constraints in its constraint store. One technique to simplify these constraints is called *constraint propagation*: The solver tries to enforce a form of local consistency on a subset of constraints in its store, which may reduce the domains of variables associated with these constraints. If the domain sizes cannot be further decreased using constraint propagation, the solver will start guessing by assigning all possible values for a variable regarding its domain. These assignments are then propagated, possibly leading to a further reduction of the variable domains. This technique is called *labeling*. Both techniques, i.e., constraint propagation and labeling, are applied iteratively until the constraint solver either finds a variable assignment satisfying all given constraints or reaches an inconsistent state, meaning that the constraints in its store are unsatisfiable. In the latter case, a form of backtracking is used to restore the last consistent state and continue the search with a different variable assignment.

FD constraint programming can be used in many application areas where scheduling or optimization problems should be solved, like work assignments or transportation scheduling. Another application of FD constraints is solving mathematical puzzles, like cryptoarithmetic puzzles where an injective assignment from symbols to digits should be found such that a given calculation becomes valid. For instance, the well-known "send-more-money" puzzle is given by the following equation:

$$SEND + MORE = MONEY$$

The goal is to assign a different decimal digit to each letter in such a way that the equation is fulfilled. Additionally, the values for the letters S and M are constrained to be greater than 0.

Using a suitable modelling language, one can specify this puzzle as an FD constraint problem and apply a constraint solver on this specification. Since we are interested in the integration of functional logic programming and finite domain constraint programming, we show a solution to this puzzle in Curry. Prolog-based implementations of functional logic languages often exploit the constraint solvers available in Prolog [AH00, FHGSP03]. For instance, PAKCS [HAB+13], a Curry implementation which translates Curry to Prolog, provides a library `CLPFD` containing the usual FD constraints and operations (standard relational and arithmetic operators are suffixed by the character "`#`" in order to distinguish them from the prelude operations on numbers). Using this library, the "send-more-money"

puzzle can be specified as follows:

```
smm xs = xs =:= [s, e, n, d, m, o, r, y]
       & domain xs 0 9
       & s ># 0  &  m ># 0
       & allDifferent xs
       &                  1000 *# s +# 100 *# e +# 10 *# n +# d
          +#              1000 *# m +# 100 *# o +# 10 *# r +# e
          =# 10000 *# m +# 1000 *# o +# 100 *# n +# 10 *# e +# y
       & labeling xs
  where s, e, n, d, m, o, r, y free
```

This model captures all the terms of the "send-more-money" puzzle representing each letter by a logic variable and connecting them with constraints like `allDifferent`. The equation is represented by arithmetic and relational constraints, and the different constraints are combined using the constraint conjunction operator "`&`". When we solve the goal `smm xs`, we obtain the single solution `xs = [9,5,6,7,1,0,8,2]` within a few milliseconds.

4 The Compilation Scheme of KiCS2

To understand the extensions described in the subsequent sections, we review the translation of Curry programs into Haskell programs as performed by KiCS2. More details about this translation scheme can be found in previous articles about KiCS2 [BHPR11, BHPR13].

Since KiCS2 compiles Curry programs into Haskell programs, the non-deterministic features of Curry must be "simulated" in some way in Haskell. For this purpose, KiCS2 represents the potential non-deterministic values of an expression in a data structure. Thus, each data type of the source program is extended by constructors to represent a choice between two values and a failure, respectively. For instance, the data type for Boolean values defined in a Curry program by

```
data Bool = False | True
```

is translated into the Haskell data type

```
data C_Bool = C_False | C_True | Choice_Bool ID C_Bool C_Bool | Fail_Bool
```

where `Fail_Bool` represents a failure and `(Choice_Bool i t1 t2)` a non-deterministic value, i.e., a selection of two values `t1` and `t2` that can be chosen by some search strategy.

The first argument `i` of type `ID` of a `Choice_Bool` constructor is used to implement the call-time choice semantics discussed in Sect. 2. Since the evaluation of `xorSelf aBool` duplicates the argument operation `aBool`, we have to ensure that both duplicates, which later evaluate to a non-deterministic choice between two values, yield either `C_True` or `C_False` consistently. This is obtained by assigning a unique identifier (of type `ID`) to each `Choice_Bool` constructor. To avoid unsafe features with side effects, KiCS2 uses the idea presented by Augustsson, Rittri and Synek [ARS94] and passes a (conceptually infinite)

set of identifiers, also called *identifier supply* (of type `IDSupply`), to each operation so that a `Choice` can pick its unique identifier from this set. For instance, the operation `aBool` defined in Sect. 2 is translated into:

```
aBool :: IDSupply → C_Bool
aBool s = Choice_Bool (thisID s) C_True C_False
```

The operation `thisID` takes some identifier from the given identifier supply. Furthermore, there are operations `leftSupply` and `rightSupply` to split an identifier supply into two disjoint subsets without the identifier obtained by `thisID`. These are necessary to provide several operations occurring in the right-hand side of a rule with disjoint identifier supplies. For instance, the operation

```
main :: Bool
main = xorSelf aBool
```

is translated into

```
main :: IDSupply → C_Bool
main s = xorSelf (aBool (leftSupply s)) (rightSupply s)
```

so that the set `s` is split into a set `(leftSupply s)` containing identifiers for the evaluation of the argument `aBool` and a set `(rightSupply s)` containing identifiers for the evaluation of the operation `xorSelf`.[1]

Note that it is not possible to assign the identifiers by a custom monad which manages a set of fresh identifiers as its internal state. If we would follow this approach, we would have to determine a fixed order in which fresh identifiers are supplied to subexpressions. If we now consider the operations

```
id x = x        loop = loop        snd (x, y) = y
```

and would decide to provide identifiers to subexpressions in the order of their occurence, for the expression "`snd (loop, id True)`" we will have to provide an identifier for `loop` first. But because `loop` is a recursive operation, we will either have to provide identifiers for the recursive calls or defer and assign them when needed. Unfortunately, the first approach requires the evaluation of `loop` and, thus, destroys laziness. For the other solution, we either cannot guarantee the uniqueness of deferred identifiers or would again require unsafe features against our intention.

Because all data types defined in the source program are extended with additional constructors to represent the potential non-determinism, one can overload these constructors using a type class. This improves readability and is also required to construct non-deterministic values of a polymorphic type. Therefore, each data type also provides an instance of the following type class:

```
class NonDet a where
  choice :: ID → a → a → a
  fail   :: a
  try    :: a → Try a
```

While the first two functions allow the generic construction of a non-derministic choice

[1] A possible implementation of `IDSupply` are unbounded integers [BHPR11], which are used in subsequent examples.

and a failure, respectively, the third function allows the generic deconstruction of non-deterministic values by representing them using the following uniform wrapper:

```
data Try a = Value a | Choice ID a a | Fail
```

Note that the data constructor `Choice` is applied to values of type `a` and not to values of type `Try a`. In consequence, applying `nondet` to a given value e only requires the evaluation of e to *head normal form*. For the Curry type `Bool`, we get the following implementation of the type class `NonDet`:

```
instance NonDet C_Bool where
  choice = Choice_Bool
  fail   = Fail_Bool

  try (Choice_Bool i a b) = Choice i a b
  try Fail_Bool           = Fail
  try x                   = Value x
```

Since all data types are extended by additional constructors, KiCS2 also extends the definition of operations performing pattern matching. For instance, the partially defined operation

```
ifTrue :: Bool → a → a
ifTrue True x = x
```

is extended by an identifier supply and further matching rules:

```
ifTrue :: NonDet a ⇒ C_Bool → a → IDSupply → a
ifTrue C_True            x _ = x
ifTrue (Choice_Bool i a b) x s = choice i (ifTrue a x s) (ifTrue b x s)
ifTrue _                 _ _ = fail
```

The second rule transforms a non-deterministic argument into a non-deterministic result and the final rule returns `fail` in all other cases, i.e., if `ifTrue` is applied to `C_False` as well as if the matching argument is already a failed computation (failure propagation). Since deterministic operations do not introduce new `Choice` constructors, `ifTrue` does not actually use the identifier supply `s` (KiCS2 analyzes such situations and removes superfluous `IDSupply` arguments).

If we apply the same transformation to the rules defining `xor` and evaluate `main`, we obtain the result

```
Choice_Bool 2 (Choice_Bool 2 C_False C_True) (Choice_Bool 2 C_True C_False)
```

Thus, the result is non-deterministic and contains three choices, where all of them have the same identifier. To extract all values from such a `Choice` structure, we have to traverse it and compute all possible choices but consider the choice identifiers to make consistent (left/right) decisions. Thus, if we select the left branch as the value of the outermost `Choice`$_{Bool}$, we also have to select the left branch in the selected argument `(Choice`$_{Bool}$ `2 C_False C_True)` so that `C_False` is the only value possible for this branch. Similarly, if we select the right branch as the value of the outermost `Choice`$_{Bool}$, we also have to select the right branch in its selected argument `(Choice`$_{Bool}$ `2 C_True C_False)`, which again yields `C_False` as the only possible value. In consequence, the unintended value `C_True` is not extracted as a result.

The requirement to make consistent decisions can be implemented by storing the decisions already made for some choices during the traversal. For this purpose, KiCS2 uses the type

```
data Decision = NoDecision | ChooseLeft | ChooseRight
```

where `NoDecision` represents the fact that the value of a choice has not been decided yet. Furthermore, there are operations to look up the current decision for a given identifier or change it, like:

```
lookupDecision :: ID → IO Decision
setDecision    :: ID → Decision → IO ()
```

Now the top-level operation that prints all values contained in a generic choice structure in a depth-first manner can be defined as follows: [2]

```
printValsDFS :: (Show a, NonDet a) ⇒ Try a → IO ()
printValsDFS (Value      v) = print v
printValsDFS Fail           = return ()
printValsDFS (Choice i a b) = lookupDecision i >>= follow
  where  follow ChooseLeft  = printValsDFS (try a)
         follow ChooseRight = printValsDFS (try b)
         follow NoDecision  = do newDecision ChooseLeft  a
                                 newDecision ChooseRight b
         newDecision d x = do setDecision i d
                              printValsDFS (try x)
                              setDecision i NoDecision
```

This operation ignores failures and prints all values not rooted by a `Choice` constructor. For a `Choice` constructor, it checks whether a decision for this identifier has already been made (note that the initial value for all identifiers is `NoDecision`). If a decision has been made for this choice, it follows the corresponding path. Otherwise, the left alternative is used and this decision is stored. After printing all values w.r.t. this decision, it is undone (like in backtracking) and the right alternative is used and stored. Besides simple depth-first-search, there are also more advanced search implementations for different strategies based on state monads which are also capable of passing a tree-like data structure of the non-deterministic results back to the user for further processing (encapsulated search) [HPR12].

In general, this operation is applied to the normal form of the main expression. The normal form computation is necessary for structured data, like lists, so that a failure or choice in some part of the data is moved to the root.

To handle logic variables, KiCS2 uses a "generator approach". Hence, logic variables are replaced by generators, i.e., operations that non-deterministically evaluate to all possible ground values of the type of the logic variable. This is justified by the fact that computing with logic variables by narrowing [Red85] and computing with generators by rewriting are equivalent, i.e., yield the same values [AH06]. For instance, the expression "`not x`", where `x` is a logic variable, is translated into "`not aBool`". The latter expression is evaluated by reducing the argument `aBool` to a choice between `True` or `False` followed

[2] The type class `Show` is required to be able to print out the computed results. As Curry does not support the concept of type classes, all data types are provided with a generated instance of `Show`.

by applying `not` to this choice. This is similar to a narrowing step [Red85] on "`not x`" that instantiates the variable `x` to `True` or `False`. Since such generators are standard non-deterministic operations, they are translated like any other operation and, therefore, do not require any additional run-time support. However, in the presence of (equational) constraints, there are more efficient methods than generating all values.

Since an equational constraint $e_1 =:= e_2$ is satisfied iff e_1 and e_2 are reducible to unifiable constructor terms, the operation "`=:=`" could be considered to be defined by the following rules (we only present the rules for the Boolean type, where `Success` denotes the only constructor of the type `Success` of constraints):

```
True  =:= True   =  Success
False =:= False  =  Success
```

Unfortunately, solving equational constraints with this implementation might result in an unnecessarily large search space. For instance, solving `x =:= y` leads to two different solutions, $\{\texttt{x} \mapsto \texttt{True}, \texttt{y} \mapsto \texttt{True}\}$ and $\{\texttt{x} \mapsto \texttt{False}, \texttt{y} \mapsto \texttt{False}\}$. Using the well-known unification principle [Rob65] as in logic programming, these different solutions can be jointly represented by the single binding $\{\texttt{x} \mapsto \texttt{y}\}$ without guessing concrete values.

To implement such bindings without side effects, KiCS2 adds *binding constraints* to computed results. These are considered by the search strategy (like `printValsDFS`) when values are extracted from choice structures. For this purpose, KiCS2 distinguishes free (logic) variables from "standard choices" (introduced by overlapping rules) in the target code by the following refined definition of the type `ID`:

```
data ID = ChoiceID Integer | FreeID Integer
```

The new constructor `FreeID` identifies a choice corresponding to a free variable, e.g., the generator for Boolean variables is redefined as

```
aBool s = Choice_Bool (FreeID (thisID s)) C_True C_False
```

If an operation is applied to a free variable and requires its value, the free variable is transformed into a standard choice by the following transformation:

```
narrow :: ID → ID
narrow (FreeID i) = ChoiceID i
narrow x          = x
```

This operation is used in narrowing steps, i.e., in all rules operating on `Choice` constructors. For instance, the second rule in the implementation of the operation `ifTrue` is replaced by

```
ifTrue (Choice_Bool i x1 x2) x s
    = choice (narrow i) (ifTrue x1 x s) (ifTrue x2 x s)
```

to ensure that the resulting choice is not considered a free variable.

The consideration of free variables is relevant in equational constraints if variable bindings should be generated. For this purpose, there is a type `EQConstraint` to represent a single *binding constraint* as a pair of a choice identifier and a decision for this identifier, and a type `Constraints` to represent complex bindings for structured data as a list of binding constraints:

```
data EQConstraint = ID :=: Decision
data Constraints  = EQC [EQConstraint]
```

Furthermore, each data type is extended by the possibility to include binding constraints:

```
data C_Bool   = ... | Guard_Bool Constraints C_Bool
```

In the same manner, the type class `NonDet` and the type `Try` are extended:

```
data Try a = ... | Guard Constraints a

class NonDet a where
  ...
  guard :: Constraints → a → a
```

Thus, `(guard` cs v`)` represents a *constrained value*, i.e., the value v is only valid if the constraints cs are consistent with the decisions previously made during search. To propagate the constraints in guarded expressions, operations that use pattern matching are extended by another rule. For instance, the operation `ifTrue` is extended by the additional rule

```
ifTrue (Guard_Bool cs e) x s = guard cs (ifTrue e x s)
```

The binding constraints are created by the equational constraint operation "`=:=`": The binding of a free variable to a constructor is represented by constraints to make the same decisions as it would be done in the successful branch of the corresponding generator operation. In case of Boolean values, this can be implemented by the following additional rules for "`=:=`":

```
Choice_Bool (FreeID i) _ _ =:= C_True
    = guard (EQC [i :=: ChooseLeft ]) C_Success
Choice_Bool (FreeID i) _ _ =:= C_False
    = guard (EQC [i :=: ChooseRight]) C_Success
```

Hence, the binding of a variable to some known value is implemented as a binding constraint for the choice identifier for this variable. However, the binding of one variable to another variable cannot be represented in this way. Instead, the information that the decisions for both variables must be identical when extracting the values is represented by a final extension to the `Decision` type:

```
data Decision = ... | BindTo ID
```

Furthermore, the definition of "`=:=`" contains the following rule so that an equational constraint between two variables yields a binding for these variables:

```
Choice_Bool (FreeID i) _ _ =:= Choice_Bool (FreeID j) _ _
  = guard (EQC [i :=: BindTo j]) C_Success
```

The consistency of constraints is later checked when values are extracted from a choice structure, e.g., by the operation `printValsDFS`. For this purpose, its definition is extended by a rule handling constrained values:

```
printValsDFS (Guard (EQC cs) x) = do
  consistent ← add cs
  if consistent then do printValsDFS (try x)
                        remove cs
                else return ()
```

The operation `add` checks the consistency of the constraints `cs` with the decisions made so far and, in case of consistency, stores the decisions made by the constraints. In this case, the constrained value is evaluated before the constraints are removed to allow backtracking. Furthermore, the operations `lookupDecision` and `setDecision` are extended to deal with bindings between two variables, i.e., they follow variable chains in case of `BindTo` constructors.

As shown in this section, KiCS2 has already some infrastructure to deal with (equational) constraints so that further constraint structures could be implemented by extending the type of constraints and implementing a specific constraint solver. However, implementing good constraint solvers for FD constraints is an expensive task. Thus, we present in the following a technique to reuse existing solvers inside KiCS2 by slightly extending the current run-time system of KiCS2.

5 Implementation of Finite Domain Constraints

The objective of this work is to support FD constraints in KiCS2. Thus, we want to implement a library `CLPFD` containing constraints and operations like:

```
(+#), (-#), (*#)                        :: Int → Int → Int
(=#), (/=#), (<#), (>#), (<=#), (>=#) :: Int → Int → Success
domain                                  :: [Int] → Int → Int → Success
allDifferent                            :: [Int] → Success
sum                                     :: [Int] → Int
...
```

To ensure compatibility with the Curry system PAKCS, we adopt its library `CLPFD` so that existing programs can be compiled using KiCS2 without further modification. Our implementation is based on the idea to collect FD constraints occurring during program execution and pass them to an external FD constraint solver when answers to a given goal should be computed. Thus, we incrementally construct the constraint model during program execution and pass it to a dedicated solver as a whole. This approach is reasonable since FD constraint programming usually consists of two phases (also shown in the "send-more-money" puzzle above): The definition of the domains and all constraints, followed by a non-deterministic labeling, i.e., variable assignment [MS98].

In order to avoid changing the standard KiCS2 compilation scheme for passing FD constraints, we exploit the fact that KiCS2 already supports equational constraints (see Sect. 4) which are collected and solved when a value of an expression should be computed. Hence, we extend the type `Constraints` to include FD constraints so that guarded expressions can also be used to pass FD constraints to the constraint solver:

```
data Constraints = ... | FDC [FDConstraint]
```

The various kinds of FD constraints are represented in Haskell by the following data types:

```
data ArithOp      = Plus | Minus | Mult
data RelOp        = Equal | Diff | Less
                  | LessEqual | Greater | GreaterEqual
```

```
data FDConstraint = FDArith   ArithOp  FDTerm FDTerm FDTerm
                  | FDRel     RelOp    FDTerm FDTerm
                  | FDDomain  [FDTerm] FDTerm FDTerm
                  | FDAllDiff [FDTerm]
                  | FDSum     [FDTerm]
                  | ...
```

Note that some of these constraints could also be expressed by simpler ones, for instance, the constraint `FDAllDiff` ts could also be expressed by combining the terms of ts pairwisely using `FDRel Diff`. However, the MCP framework provides efficient implementations for more complex constraints like `allDifferent` or `sum` so that we explicitly handle those constraints.

Furthermore, we define a type `FDTerm` to represent the arguments of FD constraints, which are either constants (integer values) or FD variables:

```
data FDTerm = Const Int | FDVar ID
```

As a unique identifier for FD variables, we reuse the `ID` type already available in KiCS2 (see Sect. 4). Thus, we can map logic variables in Curry to FD variables in Haskell using the same identifier. In fact, this representation is similar to the internal representation of Curry's `Int` type but lacks the additional constructors introduced by the compilation scheme. Consequently, we will have to deal with non-determinism before creating `FDTerm`s but gain the advantage that the `FDConstraint`s themselves are deterministic. Using these data types, we can represent an FD constraint of our `CLPFD` library as a data term of type `FDConstraint`. For instance, the constraint "`x <# 3`" is represented by the Haskell term

`FDRel Less (FDVar` $\vec{x}$`) (Const 3)`

where we denote by $\vec{x}$ the identifier (i.e., the value of type `ID`) of logic variable `x`.

Note that the constructor `FDArith`, representing binary arithmetic operations in constraints, has *three* arguments. The third argument is used to identify the result of the arithmetic operation represented by the constraint. Hence, we represent nested operations by flattening, a technique also used to implement functional (logic) languages by compiling them to Prolog [AH00, CCH06]. By flattening, we can collect all arithmetic constraints in a list wrapped by the constructor `FDC`. For instance, the constraint "`x *# (3 +# y) <# 42`" is represented at run-time in Haskell by the following list of basic constraints (where `a` and `b` are new logic variables):

```
[ FDArith Plus (Const 3) (FDVar y⃗) (FDVar a⃗)
, FDArith Mult (FDVar x⃗) (FDVar a⃗) (FDVar b⃗)
, FDRel   Less (FDVar b⃗) (Const 42) ]
```

In order to generate the constraint terms during program execution, we implement the constraints and operations of the `CLPFD` library to construct these terms. This is in contrast to Prolog-based implementations of functional logic languages where FD constraints are directly mapped into the constraints of the underlying (stateful) Prolog system [AH00, LFSH99]. For instance, the operation `(+#)` is implemented as follows:

```
(+#) :: Int → Int → Int
x +# y = ((prim_FD_plus $!! x) $!! y) xPlusY      where xPlusY free

prim_FD_plus :: Int → Int → Int → Int
prim_FD_plus external
```

The predefined Curry operator `($!!)` applies a function to the normal form[3] of the given argument. Thus, f `$!!` e first evaluates the expression e before f is applied to the normal form of e. In our case, the use of this operator is necessary to avoid passing expressions containing user-defined operations into the constraint solver. As an example, assume that `fac` is the factorial function implemented in a Curry program. Since the constraint solver has no knowledge about the implementation of this function, in a constraint like "`x +# fac 4`" the second argument is evaluated to its normal form `24` before it is passed to the constraint solver. Similarly, non-deterministic choices or failures occurring in arguments are also lifted to the level of results by applying the normal form operator.

The use of `($!!)` in the implementation of `(+#)` ensures that `prim_FD_plus` will only be applied to either integers or logic variables. Furthermore, a fresh logic variable is added as the third argument to represent the result value in the flattened representation of constraints, as discussed above. Operations marked by "`external`", like `prim_FD_plus`, are primitive or external operations, i.e., they are not compiled but defined in the code of the run-time system. To continue our example, `prim_FD_plus` is implemented in Haskell as follows:

```
prim_FD_plus :: C_Int → C_Int → C_Int → C_Int
prim_FD_plus x y xPlusY
  | isFree x || isFree y  =  Guard_Int (FDC fdcs) xPlusY
  | otherwise             =  int_plus x y
 where fdcs = [FDArith Plus (toFDTerm x) (toFDTerm y) (toFDTerm xPlusY)]
```

The implementation of `prim_FD_plus` first checks whether at least one argument is a free integer variable. In this case, a new guarded expression of type `C_Int` will be generated. This guarded expression constrains the given variable for the result, namely `xPlusY`, with the appropriate arithmetic constraint. The constraints are generated by translating the given arguments to FD terms and constructing an `FDConstraint` representing the FD addition operation using `FDArith Plus`. If both arguments of `prim_FD_plus` are ground values, i.e., integer constants, there is no need to generate a constraint, since the result can be directly computed. Hence, the constants are added via the operation `int_plus` as an optimization. The auxiliary operation `toFDTerm` maps free (integer) variables and constants into the corresponding FD structure (note that, due to the evaluation of the actual arguments to normal form, other cases cannot occur here):[4]

```
toFDTerm :: C_Int → FDTerm
toFDTerm (Choice i _ _) = FDVar i
toFDTerm v              = Const (fromCurry v)
```

[3] A Curry expression is in normal form if it does not contain any defined operation. Consequently, a logic variable also is in normal form.

[4] The overloaded function `fromCurry` maps constant values of type `C_Int` into the corresponding Haskell values of type `Int`.

The other constraints and operations of the `CLPFD` library are implemented likewise, returning guarded expressions where the guards contain the description of the constraint. In order to show a slightly larger example, we consider the classic constraint problem to place n queens on an $n \times n$ chessboard so that no queen can capture another. To keep the size of the example within reasonable limits, we consider the degenerated case $n = 2$ which, of course, has no solution. We can model this problem in Curry with the `CLPFD` library as follows:

```
twoQueens = domain [p,q] 1 2              -- domain = valid rows
          & p /=# q                       -- rows must be different
          & p /=# q +# 1  &  p /=# q -# 1 -- diagonals must be different
          & labeling [p,q]                -- labeling of FD variables
  where p, q free
```

Using our implementation, the following guarded expression is generated in Haskell when executing `twoQueens`:

```
1 Guard_Success        (FDC [FDDomain [FDVar p, FDVar q] (Const 1) (Const 2)])
2  (Guard_Success      (FDC [FDRel   Diff  (FDVar p) (FDVar q)              ])
3   (Guard_Success     (FDC [FDArith Plus  (FDVar q) (Const 1) (FDVar x)    ])
4    (Guard_Success    (FDC [FDRel   Diff  (FDVar p) (FDVar x)              ])
5     (Guard_Success   (FDC [FDArith Minus (FDVar q) (Const 1) (FDVar y)    ])
6      (Guard_Success  (FDC [FDRel   Diff  (FDVar p) (FDVar y)              ])
7       (Guard_Success (FDC [FDLabeling InOrder [FDVar p, FDVar q]          ])
8        C_Success))))))
```

As explained above, each call of a constraint relation or operation in Curry is mapped to a guarded expression in Haskell containing a data term of type `FDConstraint` representing the appropriate FD constraint. Arithmetic operations are flattened by introducing new variables (in this example denoted by $\vec{x}$ and $\vec{y}$) to which the intermediate results are bound (see lines 3 and 5). These newly introduced variables are then used as arguments of further constraints (lines 4 and 6).

Using the normal treatment of constrained values sketched in Sect. 4, we are able to thread the FD constraints throughout the evaluation. However, the above mentioned example shows that the constraints can be spread over the resulting expression, whereas typical constraint solvers, like those from the monadic constraint programming framework, require the *entire* constraint model in order to solve it. Therefore, we need a kind of preprocessing to collect all FD constraints generated during program execution into a single guarded expression. For this purpose, we introduce a function `collect` which traverses the tree-like structure of a non-deterministic expression and collects all FD constraints right before a top-level search strategy is applied.

```
collect :: NonDet a ⇒ [FDConstraint] → Try a → a
collect cs (Choice i@(FreeID _) a b) = constrain cs (choice i a b)
collect cs (Choice i            a b) = choice i (collect cs (try a))
                                                (collect cs (try b))
collect _  Fail                      = fail
collect cs (Guard (FDC c) e)         = collect  (cs ++ c) (try e)
collect cs (Guard c       e)         = guard c  (collect cs (try e))
collect cs (Value v)                 = constrain cs v
```

```
constrain :: NonDet a ⇒ [FDConstraint] → a → a
constrain cs x | null cs   = x
               | otherwise = guard (FDC cs) x

printDFS :: (Show a, NonDet a) ⇒ a → IO ()
printDFS x = printValsDFS (try (collect [] (try x)))
```

Note that the collection process generally preserves the structure of the given expression and, thus, does not influence any search strategy applied thereafter. Only guarded expressions containing FD constraints are removed and their constraints are collected. As soon as a leaf of the given expression is reached, i.e., a deterministic value or a logic variable, a new guarded expression is generated constraining the respective leaf with the collected FD constraints.[5] If the given expression contains no FD constraints, it will be left unchanged.

To pass the constraints to the external FD solver, we extend the implementation of the KiCS2 top-level search by an additional case handling guarded expressions with FD constraints. As an example, we again consider the depth-first search strategy. To cover FD constraints, its implementation is extended with the following rule:

```
printValsDFS (Guard (FDC fdcs) e) = do res ← runSolver (solver fdcs e)
                                       printValsDFS res
  where solver fdcs e = do model     ← translate fdcs
                           solutions ← solve model
                           eqcs      ← makeBindings solutions
                           return (makeChoiceStructure eqcs e)
```

The FD constraints are first converted to an internal model of the solver before this model is solved. The solutions are then converted back into equational constraints to be further processed by the regular search strategy. This is done by `makeChoiceStructure` which converts the choice between different solutions into a structure of nested `Choice`s, whereas variable assignments are converted into equational constraints. For instance, if we consider the constraints generated by

```
domain [x,y] 1 2  &  x /=# y  &  labeling [x,y]  where x, y free
```

the solver will produce the list `[[1,2],[2,1]]` of two solutions, containing the assignments for `x` and `y`, respectively. These solutions will then be transformed into the same structure that would have been generated by the following equational constraints:

```
(x =:= 1 & y =:= 2) ? (x =:= 2 & y =:= 1)  where x, y free
```

That is, the solver will generate a `Choice` with underlying `Guard` constructors containing the bindings for `x` and `y`, respectively. Consequently, an unsolvable constraint would result in an empty list of solutions, which would then be converted to the `Fail` constructor.

To conclude this section, we take a closer look at the interface integrating FD constraint solvers from the MCP framework into KiCS2. Our implementation is parametric w.r.t. different solvers, and the solver interface is provided by a Haskell type class. Basically, there are four functions which need to be implemented for a specific solver:

[5] Note that the actual implementation is slightly more evolved and avoids the quadratic run time arising from the naive use of ++.

```
class Monad solver ⇒ FDSolver solver where
  type SolverModel solver :: *
  type Solutions solver   :: *

  translate    :: [FDConstraint]     → solver (SolverModel solver)
  solve        :: SolverModel solver → solver (Solutions solver)
  makeBindings :: Solutions solver   → solver [[EQConstraint]]
  runSolver    :: solver a           → IO a
```

Since most constraint solver libraries use their own modelling language, we provide a function to translate a constraint model given by constraints of type `FDConstraint` into a semantically equivalent solver specific model. Furthermore, there is a function `solve` to apply the solver to the translated model and initiate the search for solutions. To be able to process the solutions gained by the solver, the function `makeBindings` is provided in order to generate binding constraints for the logic variables contained in the model. Finally, we need a function `runSolver` to execute the solver, which potentially may require some I/O interaction (e.g., calling a binary executable). Additionally, our interface permits a specification of the type of modelling language and the representation of solutions used by the solver using Haskell's type families.

6 Benchmarks

In this section we evaluate our implementation of finite domain constraints by some benchmarks. We consider the FD libraries of two Curry implementations, namely PAKCS (version 1.11.3, based on SICStus Prolog 4.2.3) and KiCS2 (version 0.3.0) with the Glasgow Haskell Compiler (GHC 7.6.3) as its back end and an efficient `IDSupply` implementation based on `IORef`s. All benchmarks were executed on a Linux machine running Debian Wheezy with an Intel Core i5-2400 (3.1GHz) processor and 4GB of memory. The timings were performed with the `time` command measuring the execution time (in seconds) of a compiled executable for each benchmark as a mean of three runs. The result "n/a" denotes that the execution of the benchmark was not finished within 5 minutes, while "oom" (out of memory) denotes a stack space overflow.

In the first benchmark, we consider several instances of the n-queens problem described in the previous section measuring the execution time required to compute all solutions and print them to the command line. We use the CLPFD libraries of PAKCS and KiCS2 to model the problem and compare these implementations to an implementation using a generate-and-test algorithm. Regarding KiCS2, we solve the constraint model by applying three of the solver back ends provided by the MCP framework [WS10]:

1. The Overton[6] solver, solely written in Haskell.
2. The Gecode[7] runtime solver, which uses the search transformers and strategies pro-

[6] This solver was initially written by David Overton (`http://overtond.blogspot.de/2008/07/pre.html`) and later modified by the developers of the MCP framework.

[7] `http://www.gecode.org`

vided by the MCP framework applying Gecode only for constraint propagation.

3. The Gecode search solver, which delegates both constraint propagation and labeling to Gecode applying a fixed search strategy implemented in C++.

In terms of the Overton and Gecode runtime solver, we apply MCP's depth-first search and its identity search transformer to compute all solutions. For the labeling of the FD variables we use a first fail strategy, labeling the variable with the smallest domain first. Furthermore, we utilize KiCS2's depth-first search strategy to print all solutions.

n	Gen. & Test	PAKCS	KiCS2		
			Overton	Gecode Runtime	Gecode Search
6	9.41	0.09	0.02	0.02	0.02
7	178.36	0.10	0.04	0.03	0.03
⋮					
10	n/a	0.55	2.56	0.40	0.17
11	n/a	2.25	13.10	1.64	0.60
12	n/a	11.85	74.92	8.59	3.24
13	n/a	68.16	oom	50.66	17.75
14	n/a	n/a	oom	n/a	95.94

Table 1: Run times of different instances of the n-queens problem (in seconds)

The results in Table 1 show that the naive generate-and-test approach is clearly outperformed by all other solutions that use a specific implementation of FD constraints. Considering the solver-based approaches, the Gecode solvers perform best. In particular, the Gecode search solver using a fixed search strategy implemented in C++ is nearly four times as fast as the Prolog-based implementation of PAKCS. Regarding smaller instances, the Gecode runtime solver can compete with PAKCS as well. But for larger instances, it is significantly slower than the Prolog-based approach and the Gecode search solver. The reason for the worse performance is probably related to the overhead resulting from conversions between Haskell and C++ data structures for each constraint to enable constraint propagation by Gecode. Comparing the three solver back ends provided by the MCP framework, the benchmark clearly shows that the C++-based constraint solvers outperform the Overton solver written in Haskell.

In Table 2 we compare the run time of solving four selected FD problems by either

1. using the PAKCS implementation,
2. using the MCP framework and the KiCS2 implementation of FD constraints,
3. using the Gecode search solver provided by the MCP framework, or
4. directly calling an executable[8] linked to the Gecode library.

[8]Note that we generated these executables using the code generator provided by the MCP framework.

In this benchmark, we measured the execution time required to compute and print *only the first solution*. The "grocery" benchmark searches for four prices represented in US-Cent which yield a total price of \$7.11 regardless whether they are added or multiplied, "send-more-money" is the cryptoarithmetic puzzle already described in Sect. 3 and "sudoku" computes the solution of an 9×9 Sudoku puzzle initally filled with 25 numbers. Not surprisingly, calling Gecode directly performs better than using it via the MCP framework or KiCS2. As the results show, the integration of Gecode into Haskell (via MCP) already leads to some overhead. Since our implementation of FD constraints in KiCS2 is built upon the MCP framework with a preceeding translation process similar to the one inside the MCP framework, the additional overhead indicated by the benchmarks seems reasonable. Furthermore, the results demonstrate that our implementation can compete with the Prolog-based approach of PAKCS which is outperformed by our approach in most cases.

Benchmark	Size	PAKCS	KiCS2 Gecode Gecode Search	MCP Gecode Search	Gecode binary
grocery	—	0.14	0.08	0.08	0.08
nqueens	5	0.09	0.02	0.02	0.01
	13	0.09	0.06	0.05	0.01
	27	0.44	0.66	0.17	0.03
	55	n/a	15.32	0.66	0.10
send-more-money	—	0.09	0.03	0.02	0.01
sudoku	—	0.10	0.04	0.06	0.01

Table 2: Run times of selected FD problems solved with PAKCS, KiCS2, the MCP search solver and a Gecode executable (in seconds)

7 Conclusions and Future Work

We have presented an implementation of finite domain constraints in KiCS2, a purely functional implementation of Curry. We proposed a new approach collecting all finite domain constraints occuring during a program execution and applying a functional FD constraint solver library to solve them. In order to pass the constraints to the dedicated solver, we reused KiCS2's concept of constrained values, i.e., guarded expressions. Therefore, we extended the constraint type for equational constraints to also cover the representation of FD constraints. Furthermore, we implemented a type class providing an interface to integrate functional constraint solver libraries into KiCS2. As our benchmarks demonstrate, our implementation provides a considerable performance gain compared to equational constraints alone, and can compete with or even outperform Prolog-based implementations of finite domain constraints.

For future work it might be interesting to generalize the integration of external constraint solvers in such a way that new classes of constraint systems, like SAT solvers for Boolean

constraints or solvers for arithmetic constraints over real numbers, could be integrated in KiCS2 using the same interface. Another aspect worth further investigations is the integration of an *incremental* solver like the one developed by David Overton to overcome the performance drawbacks resulting from complex intermediate constraint models and the preprocessing phase. Furthermore, this might even enable the use of the different search strategies already implemented in KiCS2 to investigate the constraint search space, thus, generalizing from backtracking to arbitrary search strategies.

References

[AH00] S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.

[AH05] S. Antoy and M. Hanus. Declarative Programming with Function Patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.

[AH06] S. Antoy and M. Hanus. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.

[AH09] S. Antoy and M. Hanus. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.

[AH10] S. Antoy and M. Hanus. Functional Logic Programming. *Communications of the ACM*, 53(4):74–85, 2010.

[AH11] S. Antoy and M. Hanus. New Functional Logic Design Patterns. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 19–34. Springer LNCS 6816, 2011.

[ARS94] L. Augustsson, M. Rittri, and D. Synek. On generating unique names. *Journal of Functional Programming*, 4(1):117–123, 1994.

[BHPR11] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.

[BHPR13] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. Implementing Equational Constraints in a Functional Language. In *Proc. of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL 2013)*, pages 125–140. Springer LNCS 7752, 2013.

[CCH06] A. Casas, D. Cabeza, and M.V. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation, and Higher-Order in LP Systems. In *Proc. of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006)*, pages 146–162. Springer LNCS 3945, 2006.

[FHGSP03] A.J. Fernández, M.T. Hortalá-González, and F. Sáenz-Pérez. Solving Combinatorial Problems with a Constraint Functional Logic Language. In *Proc. of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL 2003)*, pages 320–338. Springer LNCS 2562, 2003.

[FHGSPdVV07] A.J. Fernández, M.T. Hortalá-González, F. Sáenz-Pérez, and R. del Vado-Vírseda. Constraint Functional Logic Programming over Finite Domains. *Theory and Practice of Logic Programming*, 7(5):537–582, 2007.

[GMHGLFRA99] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

[HAB+13] M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2013.

[Han13] M. Hanus. Functional Logic Programming: From Theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.

[He12] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.

[HPR12] M. Hanus, B. Peemöller, and F. Reck. Search Strategies for Functional Logic Programming. In *Proc. of the 5th Working Conference on Programming Languages (ATPS'12)*, pages 61–74. Springer LNI 199, 2012.

[Hus92] H. Hussmann. Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting. *Journal of Logic Programming*, 12:237–255, 1992.

[JL87] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, 1987.

[LFSH99] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.

[MS98] K. Marriott and P.J. Stuckey. *Programming with Constraints*. MIT Press, 1998.

[PJ03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[Red85] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.

[Rob65] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[SSW09] T. Schrijvers, P. Stuckey, and P. Wadler. Monadic Constraint Programming. *Journal of Functional Programming*, 19(6):663–697, 2009.

[WS10] Pieter Wuille and Tom Schrijvers. Expressive models for Monadic Constraint Programming. In Toni Mancini and Justin K. Pearson, editors, *Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation,*, page 15, September 2010.