Overview of the Monadic Constraint Programming Framework

Tom Schrijvers*

Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A
3001 Heverlee, Belgium
tom.schrijvers@cs.kuleuven.be

Abstract. A constraint programming system combines two essential components: a constraint solver and a search engine. The constraint solver reasons about satisfiability of conjunctions of constraints, and the search engine controls the search for solutions by iteratively exploring a disjunctive search tree defined by the constraint program.
The Monadic Constraint Programming framework gives a monadic definition of constraint programming where the solver is defined as a monad threaded through the monadic search tree. Search and search strategies can then be defined as firstclass objects that can themselves be built or extended by composable search transformers. Search transformers give a powerful and unifying approach to viewing search in constraint programming, and the resulting constraint programming system is first class and extremely flexible.

1 Introduction

A constraint programming (CP) [11] system combines two essential components: a constraint solver and a search engine. The constraint solver reasons about conjunctions of constraints and its principal job it to determine unsatisfiability of a conjunction. The search engine controls the search for solutions by iteratively exploring an OR search tree defined by the program. Whenever the conjunction of constraints in one path defined by the search tree is unsatisfiable, search changes to explore another part of the search tree.

Constraint programming is a declarative programming formalism, where the constraints are defined declaratively, but the underlying constraint solvers are highly stateful, and indeed to specify complex search CP programs rely on reflecting state information from the solver. So in that sense constraint programming is not so declarative after all.

In the Monadic Constraint Programming (MCP) framework we give a monadic definition of constraint programming where the solver is defined as a monad threaded through a monadic search tree. We are then able to define search and search strategies as first class objects that can themselves be built or extended

* Post-doctoral researcher of the Fund for Scientific Research - Flanders.

by composable search transformers. Search transformers give a powerful and unifying approach to viewing search in constraint programming. The resulting CP system is first class and extremely flexible.

Our work can be viewed as encapsulating the functional abstractions previously used in constraint programming in a functional programming language, and using the power of functional programming to take a further step in the increasingly abstract view of search and constraint programming. The contributions of the MCP framework are:

- We show how monads provide a powerful tool for implementing constraint programming abstractions, which allows us to build a highly generic framework for constraint programming.
- We define search strategy transformers which are composable transformers of search, and show how we can understand existing search strategies as constructed from more fundamental transformers.
- We open up a huge space of exploration for search transformers.
- The code is available at `http://www.cs.kuleuven.be/~toms/Haskell/`.

The remainder of the paper is organized as follows. Section 2 provides a motivating example of the MCP framework. For those unfamiliar with Haskell type classes and monads, Section 3 introduces them briefly. Then in Sections 4, 5 & 6 the core parts of the MCP framework are presented, respectively the modeling language, the solving process and search strategies. An overview of related work is given in Section 7. Finally, Section 8 concludes.

2 Motivating Example

The n queens problem requires the placing of n queens on an $n \times n$ chessboard, so that no queen can capture another. Since queens can move vertically, horizontally, and diagonally this means that

1. No two queens share the same *column*.
2. No two queens share the same *row*.
3. No two queens share the same *diagonal*.

A standard model of the n queens problem is as follows. Since we have n queens to place in n different columns, we are sure that there is exactly one queen in each column. We can thus denote the row position of the queen in column i by the integer variable q_i. These variables are constrained to take values in the range $1..n$. This model automatically ensures the column constraint is satisfied. We can then express the row constraint as

$$\forall 1 \leq i < j \leq n : q_i \neq q_j$$

and the diagonal constraint as

$$\forall 1 \leq i < j \leq n : q_i \neq q_j + (j - i) \quad \wedge \quad q_j \neq q_i + (j - i)$$

since queens i and j, with $i < j$, are on the same descending diagonal iff $q_i = q_j + (j - i)$, and similarly they are on the same ascending diagonal iff $q_j = q_i + (j - i)$.

A solution to the 8 queens problem is shown in Figure 1. The solution illustrated has $q_1 = 8$, $q_2 = 4$, $q_3 = 1$, $q_4 = 3$, $q_5 = 6$, $q_6 = 2$, $q_7 = 7$, $q_8 = 5$.

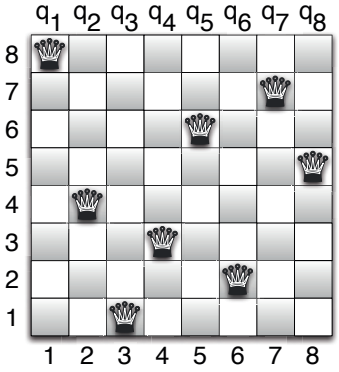

Fig. 1. A solution to the 8 queens problem

The first role of a constraint programming language is to be able to succinctly model problems. We will define constraint programming in Haskell which allows the model of the n queens problem shown in Figure 2. Note how similar it is to the mathematical model.

```
nqueens n =  exist n $ \queens -> model queens n

model queens n = queens `allin` (1,n) /\
                 alldifferent queens  /\
                 diagonals queens

allin queens range =  conj [q `in_domain` range | q <- queens ]

alldifferent queens = conj [ qi @\= qj | qi:qjs <- tails queens,
                                         qj <- qjs ]
diagonals queens =  conj [ qi @\== (qj @+ d) /\ qj @\== (qi @+ d)
                         | qi:qjs  <- tails queens, (qj,d) <- zip qjs [1..]]
```

Fig. 2. Haskell code for modelling n queens.

The next important part of a constraint programming solution is to be able to program the search. We will construct a language for search that allows us to express complex search strategies succinctly, and in a composable manner.

Search is separated into components: specifying the search tree, the basic order for visiting the search tree, and then the search transformers which transform the search tree or the way it is visited. Examples of search orders are depth-first search (`dfs`), breadth-first search (`bfs`) or best-first search. Examples of search transformers are depth bounded search (`db` n never visits nodes at depth below n), node bounded search (`nb` n visits at most n nodes), limited discrepancy search (`ld` n visits only nodes requiring at most n right branches), or branch-and-bound optimization (`bb` f applies a tree transformation f for eliminating non-optimal solutions). These search transformers are composable, so we can apply multiple transformations in order.

For example, using our search framework we can succinctly define complex search strategies. The following calls show how to solve 8 queens with:

- depth first search, first applying a node bound of 100, then a depth bound of 25, then using `newBound` branch and bound
- breadth first search, first applying a depth bound of 25, then a node bound of 100, then using `newBound` branch and bound
- breadth first search, first limited discrepancy of 10, then a node bound of 100, then using `newBound` branch and bound

can be expressed in our framework as:

```
> solve dfs (nb 100 :- db 25 :- bb newBound) $ nqueens 8
> solve bfs (db 25 :- nb 100 :- bb newBound) $ nqueens 8
> solve bfs (ld 10 :- nb 100 :- bb newBound) $ nqueens 8
```

Clearly exploring different search strategies is very straightforward.

3 Haskell Background

The MCP framework relies heavily on Haskell's type system and abstraction mechanism for providing a flexible component-based system.

We assume that the reader is already familiar with the basics of Haskell, such as algebraic data types and (higher-order) functions, but provide a brief introduction to two of the more advanced features, type classes and monads, that MCP heavily relies on.

3.1 Type Classes

Type classes [22] are Haskell's systematic solution to adhoc overloading. Conceptually, a type class C is an n-ary predicate over types that states whether an implementation of the overloaded methods associated to the type class is available for a particular combination of types.

For instance, `Eq` is a unary type class with associated method `(==)` for equality:

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

The type class constraint `Eq` τ holds if type τ provides an implementation for the method `(==)`. Implementations are provided through type class instances. For instance,

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

provides an implementation of equality for booleans. The function `(==)` has type $\forall a.$`Eq` $a \Rightarrow a \rightarrow a \rightarrow$ `Bool`. This signature expresses that `(==)` applies to argument of any type a that is an instance of the `Eq` type class. This suggests a sequential composition of computations.

Type class constraints propagate from the signature of type class methods to functions that are defined in terms of them. For instance, the function `allEqual` that checks whether all elements of a list are equal inherits the type class constraint from its use of `(==)`:

```
allEqual :: Eq a => [a] -> Bool
allEqual (x:y:zs)  = x == y && allEqual (y:zs)
allEqual _         = True
```

3.2 Monads

Monads [21] are an abstraction used in functional programming languages to represent *effectful computations*. A monad computation `m` is parametrized in the type of the computed result `a`. So `m a` denotes a monadic computation of type `m` that produces a value of type `a`. Monads are captured in the type class:

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

The `return` function creates a pure computation that simply returns a given value without any actual effects. The bind operator `(>>=)` composes two monadic computations: the first computation produces a value of type `a` that is consumed by the second computation to produce a value of type `b`. Note that the arrow `>>=` corresponds to the data flow: the left operand produces data, the right one consumes it.

Not captured in Haskell code, but also part of the monad specification, are the monad laws:

$$\begin{aligned} \texttt{return}\ x\ \texttt{>>=}\ f &\equiv f\ x \\ m\ \texttt{>>=}\ \texttt{return} &\equiv m \\ (m\ \texttt{>>=}\ f)\ \texttt{>>=}\ g &\equiv m\ \texttt{>>=}\ (\lambda x.f\ x\ \texttt{>>=}\ g) \end{aligned}$$

Any instance of the `Monad` type class should satisfy these laws, although they are not enforced by the Haskell language.

Haskell provides syntactic sugar of an imperative style for monads, the `do` notation.

$$\texttt{do}\ \begin{matrix} x\ \texttt{<-}\ m_1 \\ m_2 \end{matrix} \equiv m_1\ \texttt{>>=}\ \backslash x\ \texttt{->}\ m_2$$

$$\texttt{do}\ \begin{matrix} m_1 \\ m_2 \end{matrix} \equiv m_1\ \texttt{>>=}\ \backslash\_\ \texttt{->}\ m_2$$

There is a wide range of literature and Haskell libraries related to monad instances, and generic monad infrastructure.

4 Constraint Models

The MCP framework represents constraint models at the core by a separate data type, called `Tree`. This has obvious advantages for manipulating and inspecting the model. On top of this core data type, MCP provides convenient syntactic sugar for expressing models in a higher-level form, closer to mathematical formulas.

4.1 The Model `Tree`

The core data type for representing constraint models is defined as follows:

```
data Tree solver a
   = Return a
   | NewVar (Term solver -> Tree solver a)
   | Add (Constraint solver) (Tree solver a)
   | Try (Tree solver a) (Tree solver a)
   | Fail
   | Dynamic (solver (Tree solver a))
```

It is parametric in two types: 1) the constraint `solver` and its associated constraint domain, and 2) a computed value of type `a`. The former makes most of the constraint model infrastructure independent of the particular constraint domain, and hence reusable in many settings. The latter makes `Tree solver` a monad, and allows Haskell's monad infrastructure to be reused.

The different constructors of `Tree solver a` have the following meaning. `Return a` is a trivially satisfiable model that returns a value `a`, while `Fail` is an

inconsistent model. `Acc c m` extends a model `t` with an additional constraint `c` at the front. `NewVar f` represents a model `f` with an existentially bound (new) constraint variable. `Try t1 t2` represents a disjunctive model with the alternatives `t1` and `t2`. Finally, `Dynamic m` allows the dynamic creation of the model as a computation in the solver (see later).

Now we can make `Tree solver` a monad instance:

```
instance Monad (Tree solver) where
  return = Return

  (Return x)  >>= f  = f x
  (NewVar g)  >>= f  = NewVar (\v -> g x >>= f)
  (Add c t)   >>= f  = Add c (t >>= f)
  (Try t1 t2) >>= f  = Try (t1 >>= f) (t2 >>= f)
  Fail        >>= f  = Fail
  Dynamic m   >>= f  = Dynamic (do { t <- m ; return (t >>= f)})
```

By straightforward equational reasoning, we can establish that the monad laws hold for this monad instance. For example, the first monad law holds as follows:

$$
\begin{array}{ll}
 & \texttt{return}\ x\ \texttt{>>=}\ f \\
\equiv & \textit{(definition of } \texttt{return}\textit{)} \\
 & \texttt{Return}\ x\ \texttt{>>=}\ f \\
\equiv & \textit{(definition of } \texttt{(>>=)}\textit{)} \\
 & f\ x
\end{array}
$$

4.2 Syntactic Sugar

On top of the core data type, MCP adds various convenient abstractions:

```
true  = Return ()
false = Fail

t1 /\ t2  = t1 >>= \_ -> t2
t1 \/ t2  = Try t1 t2

conj = foldr (/\) true
disj = foldr (\/) false

exists = NewVar

exist n f = aux n []
   where aux 0 vs  = f $ reverse vs
         aux n vs  = exists $ \v -> aux (n-1) (v:vs)
```

Note that `conj` and `exist` are two domain-independent model combinators we have used in the n-queens model.

5 Constraint Solving

The constraint model presented in the previous section is a data type. In order to actually compute solutions for the model, it must be "fed" to a constraint solver. However, MCP refrains from directly exposing the original model to a constraint solver directly. Instead, MCP translates the model into a set of core primitives understood by the solver.

There are several important reasons for this approach:

- It reduces the solver implementor's burden, who must only provide core functionality for his solver.
- The MCP framework retains control over the translation process and exposes it to the framework user.
- Much of the translation logic can be reused for different constraint solvers.

5.1 The Solver Interface

The interface that constraint solvers must support is captured in the `Solver` type class:

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

First line states that a solver must be a monad. Indeed, in general we assume that the solver encapsulates a stateful computation, where the state consists of the solver's constraint store. Two associated types of the solver define its constraint domain: `Constraint solver` is the type of constraints supported by the solver, and `Term solver` is the type of terms that the constraints range over.

The two methods `newvar` and `add` are the respective counterparts of the `NewVar` and `Add` constructors of the model tree. The former returns a new constraint variable as a solver computation. The latter adds a constraint to the solver state and returns a boolean indicating whether the constraint store is still consistent (`True`) as far as the solver can tell or has become definitely inconsistent (`False`). The `run` method allows extracting the values from a solver computation.

Finally, the remaining three members of the `Solver` class are related to disjunctions. The solver interface for disjunction is much more primitive than the high-level `Try` constructor of the model. The `Label solver` type represents a label for a solver state; the label for the current solver state can be requested with

the `mark` operation. The `goto` operation restores the solver state of a given label. On top of these two operations various search strategies can be programmed.

From the side of the solver, different strategies can be used to implement the primitive operations. In a state copying approach, the labels are simply copies of the state and the operations obvious. In the case of recomputation, a label is a trace of the operations that led up to the solver state, and `goto` replays the trace. Backtracking involves a more intricate strategy.

5.2 A Simple Finite Domain Solver

To illustrate the solver interface, we present a simple instantiation, without going into the implementation details.

Our solver type is called `FD` and its instance of the `Solver` class is:

```
instance Solver FD where
  type Constraint FD  = FDConstraint
  type Term FD        = FDTerm
  newvar  = newvarFD
  ...
```

The `FDTerm` type is abstract, and of course the details of the member functions are not exposed. All the programmer needs to know are the details of the `FDConstraint` type. Our small `FD` solver only supports three constraints:

```
data FDConstraint  =  FDIn FDTerm (Int,Int)
                   |  FDEQ FDTerm Int
                   |  FDNE FDTerm FDTerm Int
```

Formally, the semantics can be expressed as:

$$\begin{aligned} [\![\texttt{FDIn } t \texttt{ (}l\texttt{,}u\texttt{)}]\!] &= [\![t]\!] \in \{l, \ldots, u\} \\ [\![\texttt{FDEQ } t \ d]\!] &= [\![t]\!] = d \\ [\![\texttt{FDNE } s \ t \ i]\!] &= [\![s]\!] \neq [\![t]\!] + i \end{aligned}$$

We use Overton's FD solver [13] for the concrete implementation.

On top of this interface, convenient syntactic sugar such as that used in the n-queens model, is easily defined.

```
x @\= y  = Add (FDNE x y 0) true
```

5.3 From Models to Solver Computations

The `eval` function turns a model into a solver computation.

```
eval :: Solver solver => Tree solver a -> solver [a]
eval model  = eval' model []
```

```
eval' (Return x) wl  = do xs <- continue wl
                          return (x:xs)
eval' (Add c t)  wl  = do b <- add c
                          if b then eval' t  wl
                               else continue wl
eval' (NewVar f) wl  = do v <- newvar
                          eval' (f v) wl
eval' (Try l r)  wl  = do now <- mark
                          eval' l ((now,r):wl)
eval' Fail       wl  = continue wl

continue []              = return []
continue ((past,t):wl)   = do goto past
                              eval' t wl
```

The `eval'` function is the main workhorse, that has a worklist of labels as an additional parameter. When a disjunction (`Try`) is encountered, the label of the current solver state is pushed onto the worklist together with the right branch for later processing, while the left branch is processed immediately. The `continue` function is invoked whenever the end of a branch is reached, or an inconsistency is detected by the solver. Then a deferred branch is popped from the worklist, its state is restored and processing continues until the worklist is empty.

Putting everything together, the list of solutions is extracted from the evaluated model with the solver's `run` method:

```
solve :: Solver solver => Tree solver a -> [a]
solve =  run . eval
```

6 Search

MCP makes search much more flexible in a number of ways, summarized in this section.

6.1 Dynamic Variable Enumeration

Often search is used to complete incomplete propagation strategies of constraint solvers. In particular, for finite domain (FD) solvers, the possible assignments for variables are enumerated in disjunctions. For instance, the following code augments the n-queens model with such enumeration.

```
nqueens n = exist n $ \queens -> model queens n /\
                                 enumerate queens [1..n]

enumerate qs values =  conj [ enum q values | q <- qs ]

enum var values = disj [ var @= value | value <- values ]
```

Note however, that this enumeration is based on the variable's static domain. The generated search tree can be much more compact, if the dynamic domain of variables, reduced by the solver's propagation, is used.

If the `FD` solver exposes a function `domain :: Term FD -> FD [Int]` to query a variable's dynamic domain, the `Dynamic` model constructor allows generating the enumeration part of the search tree dynamically.

```
nqueens n = exist n $ \queens -> model queens n /\
                                 enumerate queens

enumerate  = Dynamic . label

label []      = return ()
label (v:vs)  = do d <- domain v
                   return $ enum v d /\ enumerate vs
```

Many other dynamic enumeration strategies can be captured in a similar way.

6.2 Queueing Strategies

The `eval` function above implements depth-first search using a stack as the worklist. The MCP framework generalizes this by means of a `Queue` type class, which allows other queue-like data structures to be used to implement strategies like breadth-first search and best-first search.

```
class Queue q where
  type Elem q :: *
  emptyQ   :: q -> q
  isEmptyQ :: q -> Bool
  popQ     :: q -> (Elem q,q)
  pushQ    :: Elem q -> q -> q
```

6.3 Search Transformers

Advanced search strategies can be implemented on top of the `eval` loop and queueing strategy. For this purpose, MCP introduces the concept of *search transformers*. Examples of search transformers are various forms of pruning (node-bounded, depth-bounded, limited discrepancy), randomly flipping branches of the search tree, iterative deepening, restart optimization and branch-and-bound. MCP employs the technique of functional mixins to open the recursion of the `eval` loop and to allow a search transformer to intercept each recursive call.

In addition to basic search transformers, MCP also provides search transformer `combinators` for building advanced transformers from basic ones. The most important such combinator is the composition operator `(:-)`, which sequentially composes two transformers. For instance, the sequential composition

of a node-bounded and a depth-bounded pruner explores the seach tree up to a certain depth and up to a certain number of nodes. Another example of a combinator is an iterative restarting combinator, which generalizes both iterative deepening and restart optimization.

7 Related Work

Since our approach combines constraint and functional programming there is a broad spectrum of related work.

Constraint Programming Constraint logic programming languages allow programmable search using the builtin search of the paradigm. Each system provides predicates to define search, analogous to the `Dynamic` nodes in the model tree. For instancen, ECLiPSe [23] provides a search library which allows: user programmable variable and value selection as well as different search transformers including depth bounded search, node bounded search, limited discrepancy search, and others. One transformation cannot be applied to another, although one can change strategy for example when the depth bound finishes to another strategy. The user cannot define their own search transformers in the library, though they could be programmed from scratch.

The Oz [16] language was the first language to truly separate the definition of the disjunctive constraint model from the search strategy used to explore it [14]. Here computation spaces capture the solver state, as well as possible choices (effectively the `Dynamic` nodes). Search strategies such as DFS, BFS, LDS, Branch and Bound and Best first search are constructed by copying the computation space and committing to one of the choices in the space. Search strategies themselves are monolithic, there is no notion of search transformers.

The closest work to this paper is the search language [19] of Comet [18]. Search trees are specified using `try` and `tryall` constructs (analogous to `Try` and `Dynamic` nodes), but the actual exploration is delegated to a search controller which defines what to do when starting or ending a search, failing or adding a new choice. The representation of choices is by continuations rather than the more explicit tree representation we use. The `SearchController` class of Comet is roughly equivalent to the `Transformer` class. Complex search hybrids can be constructed by building search controllers. The Comet approach shares the same core idea as our monadic approach, to allow a threading of state through a complex traversal of the underlying search tree using functional abstractions, and using that state to control the traversal. The Comet approach does not support a notion of composable search transformers. Interestingly the Comet approach to search can also be implemented in C++ using macros and continuations [12].

Functional (Constraint) Logic Programming Several programming languages have been devoted to the integration of Functional Programming and (Constraint) Logic Programming. On the one hand, we have CLP languages with support for a functional notation of predicates, such as Mercury [17] and Ciao

[4]. MERCURY allows the user to program search strategies by using the underlying depth-first search, much like any CLP language. CIAO offers two alternative search strategies, breadth-first search and iterative deepening, in terms of depth-first search by means of program transformation.

On the other hand, we have functional programming languages extended with logic programming features (non-determinism, logical variables). The most prominent of these is the CURRY language, or language family. The PACS CURRY compiler is implemented on top of SICSTUS PROLOG and naturally offers access to its constraint solver libraries; it has a fixed search strategy. However, the KICS CURRY system, implemented in HASKELL, does not offer any constraint solvers; yet, it does provide reflective access to the program's search tree [3], allowing programmed or *encapsulated* search. As far as we can tell, their implementation technique prevents this programmed search from being combined with constraint solving.

Embedding Logic Programming in Functional Programming As far as we know, Constraint Programming has gotten very little attention from mainstream Functional Programming researchers. Most effort has gone towards the study of the related domain of Logic Programming, whose built-in unification can be seen as an equality constraint solver for Herbrand terms.

There are two aspects to Logic Programming, which can and have been studied either together or separately: logical variables and unification on the one hand and (backtracking) search on the other hand.

The former matter can be seen as providing an instance of a Herbrand term equality constraint solver for our `Solver` type class. However, it remains an open issue how to fit the works of Claessen and Ljunglöf [5] and Jansson and Jeuring [9] for adding additional type safety to solver terms into our solver-independent framework.

Logic Programming and Prolog have also inspired work on search strategies in Functional Programming. That is to say, work on Prolog's dedicated search strategy: depth-first search with backtracking. Most notable is the list-based backtracking monad—which Wadler pioneered before the introduction of monads [20]—upon which various improvements have been made, e.g. breadth-first search [15], Prolog's pruning operator *cut* [8], and fair interleaving [10].

The Alma-0 [1] has a similar objective in an imperative setting: it adds Prolog-like depth-first search and pruning features to Modula-2.

FaCiLe is a finite domain constraint library for OCaml, developed as part of the Ph.D. thesis of Nicolas Barnier [2]. FaCiLe's fixed search stratgy is depth-first search; on top of this, optimization is possible by means of both the branch-and-bound and restart strategies. The implementation relies on mutable state.

8 Conclusion and Future Work

We have given a monadic specification of constraint programming in terms of a monadic constraint solver threaded through a monadic search tree. We show how

the tree can be dynamically constructed through so called labelling methods, and the order in which the nodes are visited controlled by a search strategy. The base search strategy can be transformed by search transformers, and indeed these can be constructed as composable transformations. Our framework allows the simple specification of complex search strategies, and illustrates how complex search strategies, like branch-and-bound, or iterative deepening can be built from smaller components. It also gives great freedom to explore new search strategies and transformers, for example the optimistic branch-and-bound search.

Overall by trying to be as generic and modular as possible in defining monadic constraint programming we have a powerful tool for experimentation and understanding of search in constraint programming.

8.1 Future Work:

There are many challenges ahead of the MCP framework. To name just a few important ones: 1) to generalize our search framework to arbitrary search problems, 2) to integrate a Haskell implementation of Constraint Handling Rules [6] with the framework to provide the combination of programmable **s**earch and programmable **s**olving, and 3) to explore the performance characteristics of the framework. Currently, we are integrating the Gecode solver [7] in MCP [24].

Moreover, we think it is an important challenge for Prolog implementations to offer more flexible, programmed search strategies. The stack freezing functionality available in tabulated Prolog systems seems promising to implement the `label` and `goto` methods of the MCP framework and make this possible.

Acknowledgements

I am grateful to the collaborators on the MCP framework: Peter Stuckey, Philip Wadler and Pieter Wuille. Thanks to Christian Schulte for his feedback on the Gecode instance of the MCP framework.

References

1. Krzysztof R. Apt, Jacob Brunekreef, Vincent Partington, and Andrea Schaerf. Alma-o: an imperative language that supports declarative programming. *ACM Trans. Program. Lang. Syst.*, 20(5):1014–1066, 1998.
2. Nicolas Barnier. *Application de la programmation par contraintes à des problèmes de gestion du trafic aérien*. PhD thesis, Institut National Polytechnique de Toulouse, December 2002.
3. Bernd Brassel and Frank Huch. On a tighter integration of Functional and Logic Programming. In Zhong Shao, editor, *5th Asian Symposium on Programming Languages and Systems (APLAS'07)*, volume 4807 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 2007.
4. Amadeo Casas, Daniel Cabeza, and Manuel V. Hermenegildo. A syntactic approach to combining functional notation, lazy evaluation and higher-order in LP systems. In *8th International Symposium on Functional and Logic Programming (FLOPS'06)*, pages 146–162. Springer, 2006.

5. Koen Claessen and Peter Ljunglöf. Typed logical variables in Haskell. In *Proc. of Haskell Workshop* . ACM SIGPLAN, 2000.
6. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
7. Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.
8. Ralf Hinze. Prolog's control constructs in a functional setting - axioms and implementation. *International Journal of Foundations of Computer Science*, (12(2)):125–170, 2001.
9. Patrik Jansson and Johan Jeuring. Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, 1998.
10. Oleg Kiselyov, Chung chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). *SIGPLAN Not.*, 40(9):192–203, 2005.
11. K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.
12. Laurent Michel, Andrew See, and Pascal Van Hentenryck. High-level nondeterministic abstractions in. In Frédéric Benhamou, editor, *CP*, volume 4204 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2006.
13. David Overton. Haskell FD library. `http://overtond.blogspot.com/2008/07/pre.html`, 2008.
14. Christian Schulte. Programming constraint inference engines. In *Principles and Practice of Constraint Programming - CP97, Proceedings*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533. Springer, 1997.
15. Silvija Seres and Michael J. Spivey. Embedding Prolog into Haskell. In *Haskell Workshop'99*, Septembr 1999.
16. Gert Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer, 1995.
17. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 1996.
18. Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. MIT Press, 2005.
19. Pascal Van Hentenryck and Laurent Michel. Nondeterministic control for hybrid search. *Constraints*, 11(4):353–373, 2006.
20. Philip Wadler. How to replace failure by a list of successes. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
21. Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52, London, UK, 1995.
22. Philip Wadler and Stephen 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.
23. Mark Wallace, Stefano Novello, and Joachim Schimpf. ECLiPSe: A platform for constraint logic programming, 1997.
24. Pieter Wuille and Tom Schrijvers. The FD-MCP framework. Report CW 562, Departement of Computer Science, K.U.Leuven, August 2009.