Monadic Constraint Programming in Haskell

Calvin Houser

March 17, 2018

Abstract

We look at the structure of constraint programming and its relation to other programming paradigms. We consider the structure of a functional programming, and look at an example of managing imperative elements using monads within the pure-functional language Haskell. We discuss the implementation of a constraint solver written in Haskell, examining its use of monads to handle the state and uncertainty of a constraint solver and its solutions. We look at additional contributions to constraint programming that have built on this model. The advantages of monadic constraint programming system are considered.

1 The Structure of Constraint Programming

Constraint programming provides an understandable interface for applying computation to problems without specifying a procedure for solving them. The programmer specifies the problem as a set of constraints which restrict the possible values of variables within the problem. These constraints are then parsed by a solver, which searches the domain of each variable for a solution set of values which satisfies all the constraints. Systems can often be tasked additionally to find a set of all solutions or any number of possible solutions.

Constraint Programming systems are often restricted to or optimized for problems within some particular domain. The domain of a problem specifies the characteristics of its variables and thus its solution set. Boolean satisfiability (SAT) problems concern constraints and variables within the Boolean domain. Finite domain (FD) problems concern constraints and variables restricted to finite sets. Restricting a system to a particular domain or type of domain gives its solver a greater flexibility in optimizing its search for a solution. The scope of this article will concern problems which can be solved by a FD constraint programming system.

An common example problem for understanding constraints is the n-queens problem: given a chessboard of some dimension n x n , place n queens on the board such that no queen can capture another. The constraints for this problem can be expressed informally as the following.

1. No two queens may occupy the same column.
2. No two queens may occupy the same row.
3. No two queens may occupy the same diagonal.

A solution to which would be a set of n positions which each specify the position of a queen.

Modeling this problem formally for a constraint programming system can take advantage of its finite domain. There are n variables, each capable of being assigned a row and column position, of which there are 2^n. A standard model of this problem, as described by Schrijvers, Stuckey, and Wadler[5] simplifies these constraints using a clever approach to the labeling of each queen.

Because the chessboard has n columns, and no two queens are to occupy the same column, one queen is going to be in each column. Therefore, each queen can be identified by its column. In other words, queens $q_1 ... q_n$ are each positioned in columns 1...n. Implicitly, the first constraint is then satisfied. The following two constraints can be satisfied by assigning an integer value between 1 and n to each queen q_i to identify its row position. The following two constraints can then be expressed as follows.

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

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

Given these constraints, the solver would return a set of assignments for $q_1 ... q_n$ such that these two constraints are fulfilled, or a set of all valid assignment sets.

1.1 Structure of Constraints and Solvers

As seen in the previous example, constraints for a particular problem are stated declaratively. This structure is similar to that of logic programming languages such as PROLOG. An important part of this comparison is that the ordering of each constraint will never affect the set of possible results. Another way to consider this is that the list of constraints is stateless.

By comparison, the solver of a constraint programming system must be fully aware of its state, as described by Schrijvers, Stuckey, and Wadler[5]. This typically therefore, is implemented imperatively. The standard approach[1] to this involves modeling the domain of the problem as a tree, wherin each node represents a state of the problem, and each arc represents the valid successive assignment of a variable. Leaf nodes represent complete solutions and branch nodes represent partial solutions which may lead to a complete solution. Figure **??** shows such a tree for the n-queens problem, n equal to . This tree is then traversed by the constraint solver to find the solution set. A solution in this model is a subtree of nodes representing a single assignment to each variable and a set of arcs linking them together.

Constraint solvers generally function by either blindly building the tree and backtracking on failure, or forward-checking. Forward-checking involves restricting the domain of not-yet-assigned variables to possibilities which do not contradict the already existing assignments. Backtracking solvers generate a larger search tree, but forward-checking solvers have a higher cost associated with visiting each node[1].

An implication of this model is that following a single rule of construction, two solvers operating different traversal algorithms may obtain solutions in different orders. Further, certain problems may have tree representations wherin different traversals may be optimal for quickly finding results.

2 Functional Implementation of CP

Implementing a constraint programming system within a functional programming language is unintuitive given the limitations of controlling state within a functional paradigm. A principal notion of

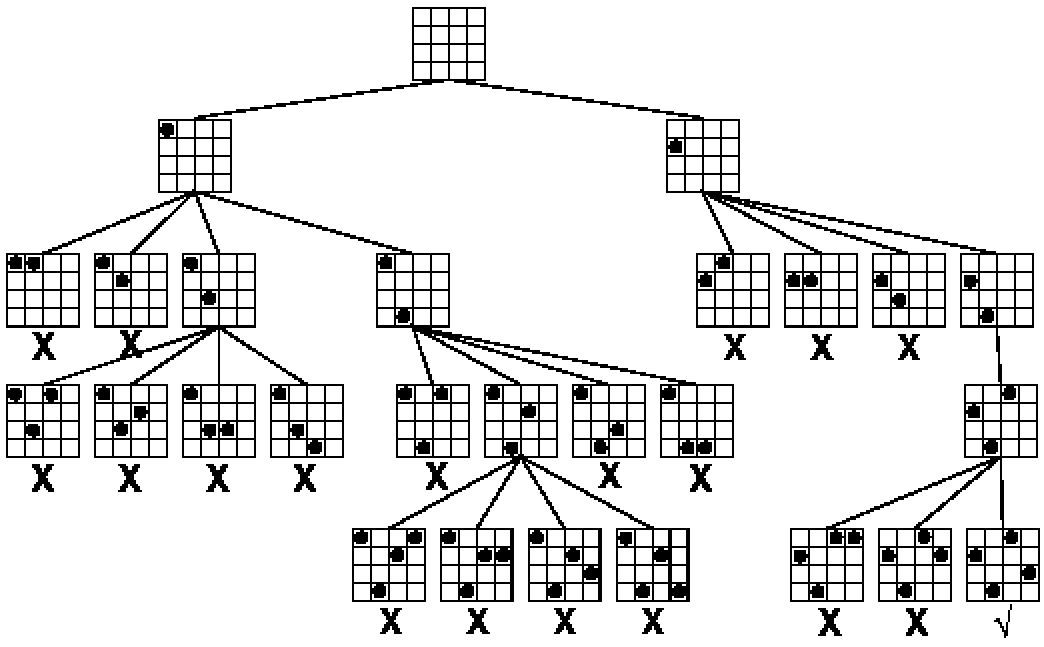

Figure 1: Tree of explored nodes for 4-queens problem

functional programming is managing computation through the evaluation of expressions or declarative functions[2], vs. as a language of procedures invoked imperatively. A series of implications from this notion inform the way a constraint programming system must be implemented.

Firstly, the result of a particular function or expression is dependant only on its input parameters. A single function, invoked in different contexts with the same parameters will always produce the same result. Consequently, a function cannot be made aware of the state in which it is run. Further, it cannot !effect! the program beyond simply returning a result.

Because a purely functional program isn't able to manipulate its state or produce effects outside of its result, functional expressions alone cannot accomplish all that of a procedurally executed program. Some imperative framework must be present to perform program output, which would typically constitute the side effect of some imperative function. Additionally, some imperative framework must further be present to invoke the current state of the program or otherwise sequence parts of its execution. Different functional languages have various ways of incorporating these necessary non-functional elements.

2.1 Haskell

The language Haskell is one of the most broadly used pure-functional programming languages in both research and industry[2].Haskell was conceived in 1987 after a proliferation of new pure, non-strict functional languages[3]. Each of these languages had established their own implemenation of non-strict, or lazy, evaluation. The difference in implementation however, prevented researchers and programmers from efficiently sharing ideas. Haskell was based principally off of two of these languages, Miranda and OL, but has made many unique adaptations to the idea of a purely functional, non-strictly evaluated functional programming language.

2.1.1 Monads and Type Classes

A particular adaptation of Haskell is the use of monads, or monadic types, to handle all imperative functionality without compromising the purely functional structure of the program. Monads conceptually originate in category theory, and are implemented as a type class in Haskell[6]. Type classes represent another idea explored uniquely within Haskell as a framework for polymorphic functions[3], and can be thought of simply as a grouping for data types that share a set of operations. Other type classes include `num` and `eq`, which group numerical and equality relations respectively.

The type class `monad` is a category of types which each allow a way of structuring computation in terms of values and sequenced computations of those values[2]. Within a Haskell program, a monad can be seen as an encapsulated imperative procedure, operating within a larger functional environment. There are three operations which define the monad type class in Haskell; the type constructor, and the `return` and `>>=` "bind" operators. These elements allow imperative procedures to be applied safely in Haskell's purely functional environment. The signatures for each operation are shown below, and enforce the safety of a monad[6].

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

        (>>)   :: m a -> m b -> m b
```

`m` is a constructor for the monad. The operation `return` takes an element `a` and encapsulates it in the monad. The operation `>>=` takes a monad with element `a` and a function. This function takes an element `a` and produces a value `b` within an instance of the monad. The resulting operation `>>=` extracts the element `a` and applies the second argument to it, flattening the monad.

The additional operator `>>` "then" has the following default implementation.

```
m >> n = m >>= \_ -> n
```

This operation, as shown by the signature and implementation, takes two monad instances and applies them sequentially without passing the result of the first into the second. Because it is defined in terms of the bind (`>>=`) operation, the default implementation can usually be used when implementing a monadic type.

2.1.2 Other Important Haskell Concepts and Notation

There are several other operators in Haskell which are important to understanding the implementation of a constraint programming system. Table 2.1.2 lists the operators which are unique to Haskell or infrequently used outside functional programming, and their uses within the Haskell CP framework.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE RankNTypes #-}
module FD (
    -- Types
    FD,           -- Monad for finite domain constraint solver
    FDVar,        -- Finite domain solver variable

    -- Functions
    runFD,        -- Run the monad and return a list of solutions.
    newVar,       -- Create a new FDVar
    newVars,      -- Create multiple FDVars
    hasValue,     -- Constrain a FDVar to a specific value
    same,         -- Constrain two FDVars to be the same
    different,    -- Constrain two FDVars to be different
    allDifferent, -- Constrain a list of FDVars to be different
    (.<.),        -- Constrain one FDVar to be less than another
    labelling     -- Backtracking search for all solutions
    ) where
```

Figure 2: David Overton's FD solver interface

`.`	Takes two functions and applies the second function to the result of the first [6]
`::`	Specifies the type of the left operand by the type expression on the right
`->`	Used in specifying type expressions, additionally the lambda definition operator; in the latter context, applies the parameters on its left to the expression on its right
`\x`	Specifies a lambda function with one or multiple parameters `x`, then followed by the defintion operator[2]
`<-`	Specifies a single assignment within the `do` monad, binding the right operand to the left operand
`=>`	Constrains the right side to take the type class specified to its left
`$`	Precedence operator, everything to its right will take precedence over anything to its left

3 Monadic Constraint Programming

The use of monads to implement a constraint solver for finite domain problems in Haskell was first explored by David Overton[4]. In his 2008 blog post, he details the implementation of an FD solver using solver monad built on top of the Haskell list monad. The resulting module interface is shown in Figure 3. A simple framework is implemented for adding variables and constraining their values, all of which are fed into the solver monad.

The solver and its associated types are defined in Figure 3. The type `FD s a` is the constraint solver, composed of a list monad wrapped in a `StateT` monad transformer. A monad transformer allows one monad to be wrapped in another, allowing the simultaneous functionality of two monadic types. In this instance, the `StateT` monad transformer encapsulates the state of each constraint

```
-- The FD monad
newtype FD s a = FD { unFD :: StateT (FDState s) [] a }
    deriving (Monad, MonadPlus, MonadState (FDState s))

-- FD variables
newtype FDVar s = FDVar { unFDVar :: Int } deriving (Ord, Eq)

type VarSupply s = FDVar s
data VarInfo s = VarInfo
     { delayedConstraints :: FD s (), values :: IntSet }
type VarMap s = Map (FDVar s) (VarInfo s)
data FDState s = FDState
     { varSupply :: VarSupply s, varMap :: VarMap s }

-- Run the FD monad and produce a lazy list of possible solutions.
runFD :: (forall s . FD s a) -> [a]
runFD fd = evalStateT (unFD fd) initState

initState :: FDState s
initState = FDState { varSupply = FDVar 0, varMap = Map.empty }
```

Figure 3: David Overton's FD solver type definitions

variable assignment, and the list monad encapsulates the set of results. The `FDState s` variable is the constraint store, holding the supply of constraint variables, and the constraints on each variable currently assigned. These constraints are represented by a set of possible values for each variable, or the variable's current domain. Changes in a variable's domain requires checking that each of its constraints are still satisfied, which can in turn limit the domain of other variables. The type variable `s` is a phantom type[6]. This concept in Haskell uses `s` to keep track of different computations on FDState. Further to the point, it keeps the monad from leaking state information into the program, the hazard in this instance being if a constraint variable from one monad was used inside of another.

The type expression for `runFD`, as shown in Figure 3, ensures that only the results of the monad are returned. Enclosing these results in a list allows the uncertainty of the problem to be enclosed within a monad. Type `list` monads in Haskell define a container type which may return any number of homogeneously-typed results, including zero (an empty list).

A complete discussion of Overton's solver can be found in ref. [4], but understanding its interface and monadic functionality is important to understanding the monadic constraint programming (MCP) framework introduced by Schrijvers et. al. [5]

3.1 The Schrijvers et. al. MCP Framework

The MCP framework, as defined by Schrijvers et. al. [5], is a fully generic solver interface, captured by a `Solver` type class. This type, shown in Figure 3.1 always includes a type representing the its type of constraints, as well as `add` and `run` functions. The `add` function takes a constraint and adds

```
class Monad s => Solver s where
    type Constraint s :: *
    add :: Constraint s -> s Bool
    run :: s a -> a
    ...
class Solver s => Term s t where
    newvar :: s t
...
data Model s a
    = Return a -- return a value
    | Add (Constraint s) (Model s a)      -- add a constraint
    | Try (Model s a) (Model s a)         -- disjunction
    | Term s t => NewVar (t -> Model s a) -- introduce new variable
    | Dynamic (s (Model s a))             -- generate subtree
```

Figure 4: MCP type classes for solvers and terms

it to the solver, and the `run` function extracts the results from the monad [7].

This model also includes a data type `Model` for the constraint model, representing a model tree. This is different from the Overton model, which modeled the constraints through a series of functions within the solver. The separate `Model` type allows a constraint problem to be specified separately from the solver and result type. The disjunctive `Try` function uses the lazy evaluation of Haskell to only construct as much of the tree as is visited[5]. This solver is first implemented as an additonal level of abstraction on the Overton solver, but later is extended to arbitrary solvers and result types[7].

3.1.1 Constraint Expressions

An improvement in using the Schrijvers et. al. MCP framework over direct use of the Overton interface is the ability to represent expressions of constraints. This is done by defining operators as syntactic sugar over the interface functions and operators to manage conjunction and disjunction. Some of these are listed in Table 3.1.1. These operators allow constraint variables to be evaluated with expressions, implemented as a binary constraint in the underlying solver.

`/\`	Conjunction operator, implemented as the `<<` operator
`\/`	Disjunction operator, implemented as a Try operation over each side of the expression
`@=`	Specifies a constraint on the left as equal to a constraint or value on the right
`@\=`	Specifies a constraint on the left as not equal to a constraint or value on the right
`@\==`	Specifies a constraint on the left as not equal to the value of a constraint on the right, plus an integer offset

As an example of this system, Figure 3.1.1 shows an implementation of the n-queens problem discussed earlier, expressed within the MCP model type. This builds the problem model, but because the evaluation is separated, the MCP framework uses a separate function which composes the `run` function of the solver with an `eval` function which parses the model. This `solve` function is shown in Figure 3.1.1.

```
exist n k = f n []
    where f 0 acc = k acc
        f n acc = NewVar $ \v -> f (n-1) (v:acc)

v ‘in_domain‘ r = Add (FDIn v r)

v1 @= n = Add (FDEQ v1 n)

data FDPlus = FDTerm :+ Int
(@+) = (:+)

v1 @\= v2 = Add (FDNE v1 v2 0)
v1 @\== (v2 :+ n) = Add (FDNE v1 v2 n)

true = Return ()
(/\) = (>>)
```

Figure 5: N-queens problem in an MCP constraint model

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

eval :: Solver solver => Tree solver a -> solver a
eval (Return x) = return x
eval (Add c t) = add c >> eval t
eval (NewVar f) = newvar >>= \v -> eval (f v)
```

Figure 6: A separate function for running the model through the solver

```
newtype LimitedDiscrepancyST = LDST Int

instance Transformer LimitedDiscrepancyST where
    type EvalState LimitedDiscrepancyST = Bool
    type TreeState LimitedDiscrepancyST = Int
    initSearch (LDST n) = (False,n)
    leftT _ ts = ts
    rightT _ ts = ts - 1
    nextT tree q t es ts
        | ts == 0 = continue q t True
        | otherwise = eval' tree q t es ts
```

Figure 7: Limited-discrepancy search transformer

3.1.2 Search Transformers

The greatest benefit to separating the model tree from the constraint solver in the MCP framework comes from transformer functions which can specify which nodes are visited in the model tree, and in what order. These are introduced by Schrijvers et. al. to specify the order in which the search tree is explored and to parametrically control the search process.

Search order is specified as either depth-first search, breadth-first search, or best first search. Ordering is managed by the evaluation function, which selects each next node for the solver from a queueing data structure. Depending on the search function, this structure functions as a stack, queue, and priority queue respectively.

Search transformers further transform the search function by adding parameters to characterize the search. The transformer sits between the evaluation function and the model-solver interaction. Here, it can impose limitations on which parts of the tree are explored. Some transformers limit the number of nodes explored or the depth of nodes explored. An example specification is shown in Figure 3.1.2 for a limited-discrepancy search transformer. This transformer will search only the leftmost path and paths including n or less right branches. Additionally, a compositional interface is created[5] for transformers, allowing any number of them to be layered together over a search.

3.2 Extending MCP

Since the original MCP framework was published, work has been done by Wuille and Schrijvers[7] extending the framework for greater compatibility with finite-domain modeling. The resulting framework, FD-MCP, is a wrapper for the MCP solver interface. FD-MCP implements a more expressive language for constraining terms, based around the structure of FD problems. Constraints can be structured more loosely, each constraint can be defined with an expression of terms on either side. Because each constraint will necessarily have a finite value, new inequality relations can be introduced (beyond those in Table **??**) which will always produce a result. Additionally for this reason, arithmetic operators are implemented and can be invoked in constraint relations.

Wuille and Schrijvers[8] further extended this framework by unifying the way boolean expressions and constraint are evaluated, allowing each to be used in place of the other. Further, constraints can be structured using standard Haskell operators. With the addition of integer arrays

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

Figure 8: N-queens modeled in FD-MCP[8]

and array expressions, higher-order Haskell functions which operate over arrays can be used in modeling FD problems. Figure 3.2 shows how the n-queens problem can be expressed in this environment. As compared to the previous implementation, it is more intuitively constructed and is easier to parse.

4 Conclusions

The advantages of using Haskell as an environment for constraint programming are based around the enforcement of its functional paradigm. When a constraint solver and modeling system can be implemented safely, the use of higher-order functions to simplify constraint modeling allows models to be much more succinct and readable.

Additional work has been done [7][8] compiling these constraint models into low-level code, allowing them to be run on different solvers. Therefore, these models can be run with the efficiency of a compiled program, using the Geocode C++ constraint solver. Further, FD-MCP is solver-independant. Overton's FD solver is still used for the native Haskell runtime, but the compilation process allows evaluation with more efficient solvers.

References

[1] Roman Barták. On-line guide to constraint programming, 1998. `http://ktiml.mff.cuni.cz/~bartak/constraints/`.

[2] HaskellWiki. The haskell programming language, 2017. `https://wiki.haskell.org/`.

[3] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM. `http://doi.acm.org/10.1145/1238844.1238856`.

[4] David Overton. Haskell fd library, 2008. `http://overtond.blogspot.com/2008/07/pre.html`.

[5] Tom Schrijvers, Peter Stuckey, and Philip Wadler. Monadic constraint programming. *Journal of Functional Programming*, 19(6):663–697, 2009.

[6] Wikibooks. Haskell, 2018. `https://en.wikibooks.org/w/index.php?title=Haskell&oldid=3370953`.

[7] Pieter Wuille and Tom Schrijvers. Monadic constraint programming with gecode. In *Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation*, pages 171–185, 2009.

[8] Pieter Wuille and Tom Schrijvers. Expressive models for monadic constraint programming. In *Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation*, page 15, 2010.