Monadic Constraint Programming in Haskell

Calvin Houser
University of California, Santa Cruz

Constraint programming

Constraint programming provides an understandable interface for applying computation to problems without specifying a procedure for solving them:

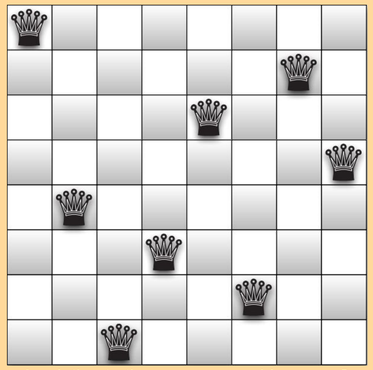
1. The programmer specifies the problem as a set of constraints which restrict the possible values of variables within the problem.
2. The solver searches the domain of each variable for a solution set of values which satisfies all the constraints.

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.

- 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.



A solution to the n-queens problem, *n* = 8

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.

- No two queens may occupy the same column.
- No two queens may occupy the same row.
- No two queens may occupy the same diagonal.

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, queens $q_1 ... q_n$ are defined as the queens in columns 1...n. 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.

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

Given these constraints, the solver would try to return a set of assignments for $q_1 ... q_n$ such that these two constraints are fulfilled.

Functional Programming

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 functional programming is managing computation through the evaluation of expressions or declarative functions. A series of implications from this notion inform the way a constraint programming system must be implemented.

- ➔ 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.
- ➔ A function cannot be made aware of the state in which it is run.
- ➔ A function cannot affect the program beyond simply returning a 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.
- Some imperative framework must further be present to invoke the current state of the program or otherwise sequence parts of its execution.

Haskell

`Monad` is a type class in Haskell which allows a way of structuring computation in terms of values and sequenced computations of those values[1]. Within a Haskell program, a monad can be seen as an encapsulated imperative procedure, operating within a larger functional environment.

```
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 \verb!a! and applies the second argument to it, flattening the monad.

References

[1] HaskellWiki. The haskell programming language, 2017. https://wiki.haskell.org/.
[2] David Overton. Haskell fd library, 2008.http://overtond.blogspot.com/2008/07/pre.html.
[3] Tom Schrijvers, Peter Stuckey, and Philip Wadler. Monadic constraint programming. Journal of Functional Programming, 19(6):663–697, 2009.
[4] 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.
[5] 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.

Overton's FD Solver

The use of monads to implement a constraint solver for finite domain problems in Haskell was first explored by David Overton[2]. A simple framework is implemented for adding variables and constraining their values, all of which are fed into the solver monad.

```
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
```

Monadic Constraint Programming

The monadic constraint programming (MCP) framework, as defined by Schrijvers et. al.[3], is a fully generic solver interface, captured by a `Solver` type class.

This model also includes a data type `Model` for the constraint model, representing a model tree. The separate `Model` type allows a constraint problem to be specified separately from the solver and result type.

Search transformers direct 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. Transformers can, for example, limit the number of nodes explored or the depth of nodes explored.

FD-MCP

Wrapper for the MCP solver interface for finite domain problems.[4][5]

- Constraints can be defined in expressions.
- Constraints can be structured using standard Haskell operators.
- Integer arrays and array expressions for variables and constraints.
- Constraints can be defined using higher-order Haskell functions.
- Models can be compiled to be run with the C++ solver Geocode

Conclusions

The advantages of using Haskell as an environment for constraint programming are based around the enforcement of its functional paradigm. The use of higher-order functions simplifies constraint modeling.

Additional work has been done compiling these constraint models into low-level code, 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.