Under consideration for publication in J. Functional Programming

Monadic Constraint Programming

TOM SCHRIJVERS
Department of Computer Science, K.U.Leuven, Belgium
PETER STUCKEY
National ICT Australia, Victoria Laboratory
Department of Computer Science and Software Engineering
University of Melbourne, Australia
PHILIP WADLER
School of Informatics, University of Edinburgh, UK

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. In this paper we give a monadic definition of constraint programming where the solver is defined as a monad threaded through the monadic search tree. We are then able to define search and search strategies as first class 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) (Marriott & Stuckey, 1998) 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 this paper 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 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.

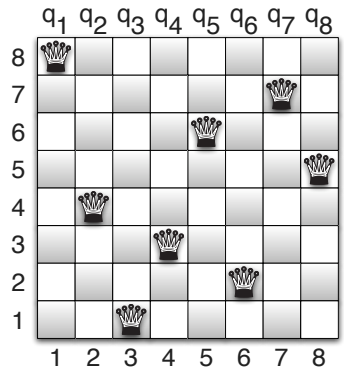

Fig. 1. A solution to the 8 queens problem

Example 1 We use the well known n queens problem as a running example throughout this paper. 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 $q_1 = 8, q_2 = 4, q_3 = 1, q_4 = 3, q_5 = 6, q_6 = 2, q_7 = 7, q_8 = 5$.

□

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.

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

```
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..]]

conj = foldl (/\) true
```

Fig. 2. Haskell code for modelling n queens.

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.

Example 2 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. □

We hope that his paper will illustrate to the functional programming community how the abstractions and mechanisms from functional programming such as monads, higher-order functions, continuations and lazy evaluation are valuable notions for defining and building constraint programming systems.

Functional abstractions have been used throughout the development of constraint programming. Indeed in the evolution of constraint programming we have seen a series of increasingly complex abstractions as the field became better understood. The original

CLP(X) framework (Jaffar & Lassez, 1987) already abstracted the constraint language and solver from the rest of the language. CLP languages such as ECLiPSe (ECLiPSe, 2008) provided search abstractions that allowed features of search such as variable and value selection to be defined by users code. The Oz (Smolka, 1995) language separated the definition of the search tree, from the method used to explore it, providing functional abstractions for defining the search. OPL (Van Hentenryck *et al.*, 2000) provided for the first time search strategy transformers in the form of search limiters, and mechanisms to define these transformers. Most recently Comet (Van Hentenryck & Michel, 2006) defined search through continuations and provided functional abstractions to thread complex state through a search in order to build complex search strategies.

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 this paper 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. In Section 2 we give a generic definition of a constraint solver and show how it can be instantiated by a simple Finite Domain (FD) solver. In Section 3 we define a structure for representing conjunctive constraint models. In Section 4 we extend the structure to model disjunctive constraint models, and now have sufficient expressiveness to run a model for the first time. In Section 5 we extend the modelling structure to allow dynamic definitions of the model, and are now able to define a basic CP system, with user defined search. In Section 6 we introduce basic search strategies which define the order in which nodes in the dynamic search tree are visited. In Section 7 we introduce search strategy transformers, which allow us to modify a given search as it proceeds. In Section 8 we extend search strategy transformers to be composable, so that we can apply multiple transformations to an underlying search. In Section 9 we discuss related work, and in Section 10 we conclude and discuss future work.

2 Constraint Solvers

The core component of a constraint programming system is the constraint solver. This is the engine that determines if conjunctions of constraints may be satisfiable, and can reflect information that it knows about a conjunction of constraints.

We define a fully generic constraint solver interface in what follows, and then we show how this interface can be instantiated for our finite domain constraint solver for n queens.

2.1 Generic Solver Interface

Our generic constraint solver interface is captured by 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
```

This type class requires that `solver` be a monad, which allows it to encapsulate its state. A solver has two associated types (Schrijvers *et al.*, 2008), the type of supported constraints (`Constraint solver`) and the type of terms that constraints range over (`Term solver`).

The terms of interest are of course the constraint variables; a fresh variable is generated by the `newvar` function. Constraints over terms are added to the solver's current set of constraints (its state) by the `add` function. The function also returns whether this addition leads to a possibly consistent set of constraints—it returns `False` if the solver knows that the conjunction of constraints is inconsistent. As this is the only operation that may *fail*, it is undesirable to require that `solver` is an error monad. Resorting to a monad transformer to indicate the potential failure of `add`, e.g., by giving it signature `Constraint solver -> MaybeT solver ()`, strikes us as overkill too.

Finally, the `run` function allows us to run an action in the solver monad and get back a result.

2.2 A Simple Finite Domain Solver

To illustrate the above generic solver interface, we now 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
  add     = addFD
  run     = runFD
```

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
```

The first, `FDIn`, restricts a variable to a range, the second, `FDEQ`, forces a variable to take a value, and the third, `FDNE`, expresses that a variable is not equal to the sum of another variable and an integer. Formally, the semantics can be expressed as:

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

We use Overton's FD solver (Overton, 2008) for the concrete implementation.

The above is sufficient to express our n-queens model. However, it is a rather primitive way of writing a model, directly against the constraint solver interface. In the next section, we define a more suitable generic model abstraction that is further away from the constraint solver interface.

Note that global constraints, such as `alldifferent` (Régin, 1994), an important feature of constraint programming, may be supported directly by the solver by making them part of the `Constraint solver` type. They can also be defined succinctly by decomposition to more primitive constraints (as we do for `alldifferent` in Figure 2) using the primitive solver interface.

3 Model Tree

We wish to represent the constraint model as a separate data type rather than a composition of functions from the constraint solver interface. This has many obvious advantages for manipulating and inspecting the model, which will come in handy later-on.

Although right now our model is only a heterogeneous sequence, we call the model data type `Tree`.

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

This data type has the two obvious constructors `NewVar` and `Add` that mimic the corresponding functions from the solver interface. Finally, the `Return` constructor is the base case and marks the end of a model.

The type `Tree` has two type parameters: `solver` is the obvious constraint solver type, while `a` is a value returned in the `Return` node. It turns the `Tree` type into its own free monad:

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

(Return x) `bind` k  = k x
(NewVar f) `bind` k  = NewVar (\v -> f v `bind` k)
(Add c  t) `bind` k  = Add c (t `bind` k)
```

Observe that `bind` effectively extends a model by replacing the base case `Return` by another model.

In terms of this new model data type, we can express the auxiliary functions of the n-queens problem as well. They are shown in Figure 3. `exist` n k creates n new variables

```
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 ()
(/\)               = (>>)
```

Fig. 3. Solver interface as `Tree` generator for n queens.

passed as an argument to k, v `in_domain` r constrains variable v to be in range r, `@=`, `@\=` and `@\==` are syntactic sugar for the underlying constraint of equality, disequality and disequality with offset, `true` represents the always true constraint, and conjunction `/\` is simply the monadic `>>` bind operation.

Because the model itself now does not run the problem through the solver, we must provide a separate "evaluation" function:

```
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)
```

4 Disjunctive Model Tree

Finite domain solvers, as well as most other constraint solvers, are incomplete; that is when given a conjunction of constraints they can give three possible answers: satisfiable, unsatisfiable or unknown (when the solver cannot determine if the conjunction is satisfiable or not). In order to *complete* a constraint solver and find one or more solutions, additional constraints must be added until the solver establishes a solution—or inconsistency. Of course, it is not a priori known what constraints need to be added in order to find a solution; one must *try* out different alternatives.

For this purpose we extend the model tree data type with two new constructors:

```
data Tree solver a
   = ...
   | Try (Tree solver a) (Tree solver a)
   | Fail
```

The `Try` constructor denotes two disjoint alternatives, for exploring the space of candidate solutions. The `Fail` constructor denotes a dead end. The `bind` function is extended in the obvious manner to cover these new constructors:

```
Fail      `bind` k = Fail
(Try l r) `bind` k = Try (l `bind` k) (r `bind` k)
```

Note that the binary `Try` constructor could be generalized to an arbitrary number of alternatives, giving it the signature `Try :: [Tree solver a] -> Tree solver a`. For the simplicity of the presentation, we avoid doing so. Moreover, no expressiveness is lost as variadic disjunctions are easily decomposed into binary disjunctions.

4.1 Solving with Branches: Solver State Snapshots

We need to extend our solver to handle `Try` nodes. In order to do so we make a minimal extension to the constraint solver interface:

```
class Monad solver => Solver solver where
  ...
  type Label solver :: *
  mark :: solver (Label solver)
  goto :: Label solver -> solver ()
```

The solver must now support a "label" type, that represents a solver state in one way or another. The `mark` function returns a label for the current solver state, and the `goto` function brings the solver back into an earlier state.

It is up to the solver to choose a representation for its labels and what strategy to use for returning to a previous state from the current one. Two common techniques are *copying* and *trailing*. In the former approach, a label is a copy of the whole solver state and the two functions are obvious. A copying solver could also use adaptive recomputation (Schulte, 1999).

In the latter approach, a label is a trail of incremental changes. Navigating from the current state to a previous state happens as follows:

- We determine all common incremental changes between the trails of the two states, starting at the root.
- All other changes of the current state are undone.
- All other changes of the previous state are redone.

The idea is that the incremental changes are cheaper to undo and redo than actual full recomputation.

Now the solving strategy goes down one branch, but remembers to revisit the other branch: the branch is pushed onto a worklist together with a label for the current solver state. The code for `solve` is shown in Figure 4. Here the worklist is a stack and the search is implicitly depth-first left to right.

Example 3 With these new constructors, we can now extend the *n*-queens model to enumerate the possible values of each variable in different disjunctions. The resulting code is

```
solve  = run . eval

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
```

Fig. 4. Code for `solve` showing how a model tree is evaluated by threading a solver along its branches.

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

enumerate queens values =  conj [ enum queen values | queen <- queens ]

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

disj = foldl (\/) false

(\/)  = Try
false = Fail
```

Fig. 5. Model for n queens with explicit enumeration of variables values.

shown in Figure 5. The `enumerate` function creates a conjunction of enumerations using `enum var values` which creates a disjunction setting the variable to each of the given values. Note how disjunction `\/` is simply `Try` and the `false` constraint is `Fail`.

The `enumerate` function constructs a tree of disjunctive constraints, for example for 2 queens it constructs the tree shown in Figure 6 which is represented by the term

```
Try (Add (q1 @= 1)                                  -- 1
         (Try (Add (q2 @= 1)                        -- 2
                   Return ())                       -- 4
              (Try (Add (q2 @= 2)
                        Return ())                  -- 5
                   Fail))
    (Try (Add (q1 @= 2)
```

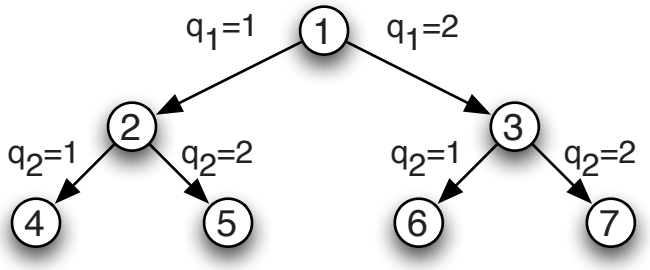

Fig. 6. Enumeration tree for 2-queens.

```
        (Try (Add (q2 @= 1)                    -- 3
             Return ())                        -- 6
             (Try (Add (q2 @= 2)
                       Return ())              -- 7
                  Fail)))
    Fail))
```

We can now run the code:

```
> solve (nqueens 1)
[()]
> solve (nqueens 2)
[]
> solve (nqueens 3)
[]
> solve (nqueens 4)
[(),()]
```

Each `()` indicates a solution. □

As the example above shows, running the solver is not very informative. The result tells us there are two solutions, but does not tell us what they actually are. Yet, what we are really interested in are of course the actual solutions.

One way to return the actual solutions is to modify the `enumerate` function and make it generate a `Return` leaf that lists the assignments made. However, in the next section we will see a much more concise way to achieve the same effect.

5 Dynamic Model Tree

The full n-queens model tree has an overwhelming n^n leaves as well as a proportional number of internal `Try` nodes. Even though lazy evaluation will avoid constructing all of this tree, many of the choices immediately lead to failure. We can do much more efficient and interesting search if we can make use of *dynamic* information in the solver.

In this section, we show several techniques for dynamically building a smaller tree, based on the solver state. Importantly we only build the tree for the parts of the search space we actually explore. For this purpose, we add one more constructor to the tree data type:

```
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
```

Fig. 7. Code for enumerating the search for n queens.

```
data Tree solver a
   = ...
   | Dynamic (solver (Tree solver a))
```

The evaluation function deals with this new constructor as follows:

```
eval' (Dynamic m) wl  = do t <- m
                           eval' t wl
```

Note that the `Dynamic m` adds additional expressiveness because the generated tree is not fixed statically, but depends on the dynamic solver state.

5.1 Branching on the Dynamic Range

Earlier, we branched the n-queens model on the full static range $(1,n)$ of each variable. However, many of the values in this range will directly result in inconsistency. Indeed, finite domain constraint solvers keep track of each variable's set of possible values. This domain dynamically shrinks as new constraints are added. By the time that possible values are enumerated a queen's domain has shrunk to a set $d \subseteq \{1..n\}$. Any attempt to assign a value v where $v \notin d$ is in vain. By dynamically inspecting the variable's current domain, we can avoid generating these redundant attempts.

Example 4 We assume that the FD constraint solver exposes a function `domain :: FDTerm -> FD [Int]` to inspect a variable's current domain. Then we can dynamically generate the enumeration subtree, based on each variable's dynamic domain. The code is shown in Figure 7.

□

5.2 Variable and Value Selection

So far we have selected the queen variables for value assignment in their natural order. However, there is no particular semantic reason to do so: any order yields the same solutions, though perhaps in a different order. However, the size of the search tree may be quite different. It turns out that for n-queens we get a much smaller search tree, if we select the variable with the smallest domain first; this is called the *first-fail* principle. The idea behind first-fail is that straining the bottleneck, the variable with the smallest degree of freedom, exposes inconsistency more quickly.

```
enumerate vs  = Dynamic . (label firstfail middleout vs)

label varsel valsel vs = do vs' <- varsel vs
                            label' vs'
  where label' []      = return ()
        label' (v:vs)  = do d <- valsel $ domain v
                            return $ enum v d /\
                            Dynamic . (label varsel valsel vs)

firstfail vs  = do ds <- mapM domain vs
                   return [ v | (d,v) <- zip ds vs
                              , then sortWith by (length d) ]
middleout l = let n = (length l) `div` 2 in
              interleave (drop n l) (reverse $ take n l)

interleave []     ys = ys
interleave (x:xs) ys = x:interleave ys xs
```

Fig. 8. Code for generic labelling predicate and examples of first fail variable selection and middleout value ordering.

Similarly we can order the way in which we try values for each queen. This does not change the size of the search tree, but may push the solutions to the left of the search tree so in left-to-right search they are found earlier. For the *n*-queens problem it is known that trying the values of a variable from the middle of the board outwards is beneficial.

Figure 8 gives generic code for a labelling function that takes two arguments: `varsel` reorders a list of variables so that the selected variable is first, while `valsel` reorders a list of values so that they are tried in the left to right order. The figure also shows implementations of first-fail variable selection and middleout value ordering. First-fail variable selection chooses the variable with least domain size. Middleout ordering tries values closest to the middle of the domain first, implemented by splitting the list in half and interleaving the reverse of the first half with the second half.

5.3 Running the Solver Revisited

If the FD solver exposes a function `value :: FDTerm -> FD Int` that returns the value assignment of a variable, the `Dynamic` constructor allows us to return the solution in a highly convenient way:

Example 5 Let's extend our *n*-queens code, so that the solutions are returned. We simply need to capture the assignments of the variables at a solution, and return them. The code is shown in Figure 9. Now, running the solver, we get to see the actual solutions:

```
> solve (nqueens 1)
[[1]]
> solve (nqueens 2)
[]
> solve (nqueens 3)
```

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

assignments   = mapM assignment
assignment q  = Dynamic $ value q >>= (return . Return)
```

Fig. 9. Code to output the solutions to an *n* queens problem.

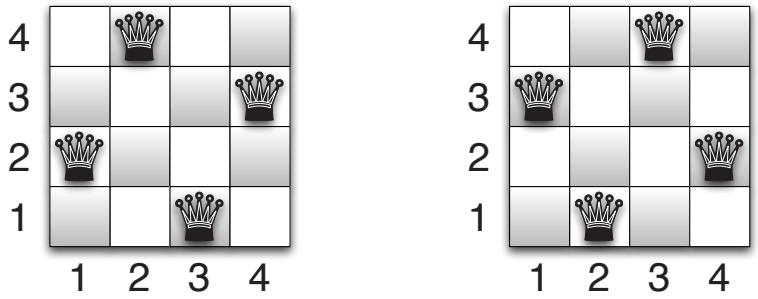

Fig. 10. The two solutions to the 4-queens problem.

```
[]
> solve (nqueens 4)
[[2,4,1,3],[3,1,4,2]]
```

The two solutions to the four queens problem are shown in Figure 10. □

5.4 Alternate Labelling Strategies

Being able to flexibly define a dynamic model tree is a key capability for constraint programming. To show the importance of the right labelling approach we compare some different approaches.

Table 1 compares the number of nodes visited to find all solutions, or just the first solution, for 4 different strategies:

- in order, the default approach defined in Figure 7,
- first fail (ff) using the code of Figure 8 with `enumerate = Dynamic . (label firstfail id)`,
- ff + middle out using the code of Figure 8, and
- ends out where variables are selected alternately from each end of the board (a bad search order).

Clearly first fail significantly improves upon in order, while ff + middle out usually improves again for finding the first solution (for all solutions it is identical to first fail). The ends out search approach is very bad.

Table 1. *Table comparing the number of Try nodes visited to find All or the First solution, using various labelling strategies. — indicates more than 100,000 nodes.*

n	in order		first fail		ff + middle out		ends out	
	All	First	All	First	All	First	All	First
1	9	8	9	8	9	8	9	8
2	13	13	13	13	13	13	13	13
3	23	23	23	23	23	23	23	23
4	84	52	84	52	84	44	84	52
5	282	62	280	62	280	62	288	64
6	329	117	325	113	325	103	369	139
7	1,531	107	1,479	107	1,479	109	1,683	129
8	4,378	254	4,286	252	4,286	138	5,166	228
9	17,496	190	16,756	186	16,756	166	21,318	290
10	54,141	285	52,371	221	52,371	205	73,515	423
11	—	261	—	293	—	259	—	839
12	—	514	—	494	—	276	—	566
13	—	370	—	544	—	354	—	1,848
14	—	1,995	—	501	—	375	—	2,487
15	—	1,555	—	421	—	473	—	4,465
16	—	9,738	—	480	—	518	—	33,418
17	—	6,430	—	584	—	544	—	17,070
18	—	37,797	—	665	—	683	—	—
19	—	3,631	—	761	—	647	—	—
20	—	—	—	748	—	830	—	—
21	—	—	—	954	—	1,358	—	—
22	—	—	—	845	—	861	—	—
23	—	—	—	901	—	905	—	—
24	—	—	—	1,008	—	996	—	—
25	—	—	—	1,082	—	1,534	—	—
26	—	—	—	1,137	—	1,195	—	—
27	—	—	—	1,219	—	1,251	—	—
28	—	—	—	1,536	—	1,362	—	—
29	—	—	—	1,664	—	1,532	—	—
30	—	—	—	1,893	—	1,525	—	—
31	—	—	—	1,679	—	1,817	—	—
32	—	—	—	1,800	—	3,390	—	—
33	—	—	—	5,670	—	2,060	—	—
34	—	—	—	2,081	—	2,529	—	—
35	—	—	—	2,309	—	2,057	—	—
36	—	—	—	2,194	—	2,610	—	—
37	—	—	—	2,372	—	2,272	—	—
38	—	—	—	2,443	—	2,985	—	—
39	—	—	—	2,745	—	2,655	—	—
40	—	—	—	3,492	—	2,608	—	—

6 Search Strategies

The way our evaluation function is implemented it visits the nodes in the tree in a depth-first order, i.e. realizing depth-first search (DFS). DFS is not always the best search strategy, if we want to get the first solution as quickly as possible. For instance, if the solution resides far to the right in the tree, DFS will find it late in the search. Then breadth-first search

(BFS) may be a better choice. Another approach is best-first search (BeFS) which orders the nodes to visit based on a heuristically determined priority.

It is common folklore that all of these search strategies are instances of a single primitive search strategy that is parametric in the queuing data type. We capture the generic queuing data type interface in the `Queue` type class:

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

where `Elem q` is the type of the elements in the queue, and the obvious functions are supported.

The evaluation function is adapted to use this `Queue` interface:

```
eval' :: (Solver solver, Queue q, Elem q ~ (Label solver,Tree solver a))
      => Tree solver a -> q -> solver [a]
eval' (Return x) wl  = ...
...
eval' (Try l r)  wl  = do now <- mark
                          continue $ pushQ (now,l) $ pushQ (now,r) wl
...

continue wl  | isEmptyQ wl  = return []
             | otherwise    = let ((past,t), wl') = popQ wl
                              in do goto past
                                    eval' t wl'
```

By choosing a LIFO queue (a stack), a FIFO queue or a priority queue we obtain respectively DFS, BFS and BeFS:

```
instance Queue [a] where
  type Elem [a] = a
  emptyQ _     = []
  isEmptyQ     = Prelude.null
  popQ (x:xs)  = (x,xs)
  pushQ        = (:)

instance Queue (Data.Sequence.Seq a) where
  type Elem (Data.Sequence.Seq a)  = a
  emptyQ _                    = Data.Sequence.empty
  isEmptyQ                    = Data.Sequence.null
  popQ (Data.Sequence.viewl -> x Data.Sequence.:< xs)  = (x,xs)
  pushQ                       = flip (Data.Sequence.|>)

solveDFS = run . eval []
```

```
solveBFS = run . eval Data.Sequence.empty

eval = flip eval'
```

7 Search Strategy Transformers

Many more search strategies can be expressed naturally in terms of *transformations* of the above primitive queue-based search strategy. For example, node-bounded search only explores a certain number of nodes, depth-bounded search only explores nodes up to a certain depth, limited-discrepancy limits the number of right branches, ...

Hence, we introduce a type class of `SearchTransformers`.

```
class Transformer t where
  type EvalState t :: *
  type TreeState t :: *
  ...
```

A search transformer `t` has its own state to base decisions for steering the search on. This state is composed of two separate parts: the `EvalState t` is threaded through the evaluation from one visited tree node to the next, while the `TreeState t` is threaded top-down through the tree from a parent tree node to its children. The usefulness of these two state components will become clear when we consider specific search transformer instances.

The search transformer acts as an extra layer on top of the primitive search. Whenever a new node is selected for processing, the search transformer gets to see it first, and, when it's done, the transformer delegates the node to the primitive search.

This means that we have to modify the code of the evaluation function to carry around the transformer and its state, and to call the transformer at the appropriate times.

```
type SearchSig solver q t a =
     (Solver solver, Queue q, Transformer t,
          Elem q ~ (Label solver,Tree solver a,TreeState t))
     => Tree solver a -> q -> t -> EvalState t -> TreeState t -> solver [a]

eval' :: SearchSig solver q t a
```

The two fundamental changes to the code are in the `Try` case of `eval'` and in `continue`.

```
eval' (Try l r)  wl t es ts  =
  do now <- mark
     let wl' = pushQ (now,l,leftT  t ts) $ pushQ (now,r,rightT t ts) wl
     continue wl' t es

continue wl t es
        | isEmptyQ wl  = return []
        | otherwise    = let ((past,tree,ts),wl') = popQ wl
                         in  do goto past
                                nextT tree wl' t es ts
```

The new functions `leftT`, `rightT` and `nextT` are methods of the `Transformer` type class:

```
leftT, rightT :: t -> TreeState t -> TreeState t
leftT  _  = id
rightT    = leftT
nextT :: SearchSig solver q t a
nextT  = eval'
...
```

With `leftT` and `rightT`, the transformer specifies how the tree state is inherited from a `Try` node. The default implementation for the right child does the same as the left child, and the latter simply copies the parent's node state.

The `nextT` method is a proxy for `eval'`, and by default it simply calls `eval'`. Note that they have the same signature.

In order to start a new search, we provide the main evaluation function:

```
eval tree q t  = let (es,ts) = initT t
                 in eval' tree q t es ts
```

which uses the last method of the `Transformer` class, for initializing the global and node states.

```
initT :: t -> (EvalState t,TreeState t)
```

7.1 Transformer Implementations

Now that we have the infrastructure, let us look at a few concrete search transformers.

Depth-Bounded Search Transformer In depth-bounded search, we do not explore any nodes beyond a certain depth in the tree. For this purpose, the tree state represents the depth of the node and the eval state simply records if any depth pruning occurred (this will be useful later).

```
newtype DepthBoundedST = DBST Int

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

The initial depth limit is embedded in the `DepthBoundedST` value. We see in `leftT` that each time a left (and through defaulting also right) branch is taken, the limit decreases. When the limit hits 0, then `nextT` does not continue evaluation at the current node, and changes the eval state to `True` to indicate pruning. This effectively cuts off the model tree at the given depth, fully orthogonal to the queuing strategy.

Node-Bounded Search Transformer In node-bounded search, at most n nodes in the tree are explored. Hence, its eval state `Int` denotes the number of remaining nodes and the tree state is unused.[1]

```
newtype NodeBoundedST = NBST Int

instance Transformer NodeBoundedST where
        type EvalState NodeBoundedST = Int
        type TreeState NodeBoundedST = ()
        initT (NBST n)  = (n,())
        nextT tree q t es ts
          | es == 0    = return []
          | otherwise  = eval' tree q t (es - 1) ts
```

The implementation of the node-bounded search transformer is almost identical to that of the depth-bounded search transformer. However, the subtle interchange of the roles of the two states, has quite a different impact on where the tree is cut off. Unlike the previous transformer, the impact of the current transformer is sensitive to the underlying queuing strategy.

Limited Discrepancy Search Transformer In limited discrepancy search, the left-most path in the tree is visited, and any other path that deviates at most n alternatives from the left-most path.

```
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
```

The effect of the above three search transformers is sketched in Figure 11.

7.2 Solver-Dependent Transformers

The `Transformer` type class forces the transformer to be fully independent of the constraint solver used. However, in some cases we want the transformer to be aware of it. For instance, assume that the FD solver keeps track of the number of times constraints are woken up and reconsidered for propagation, which can be queried with `wakeUps :: FD`

[1] We use Haskell's unit type `()` to denote empty state.

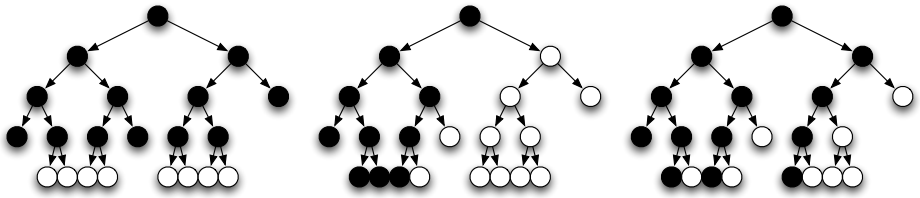

Fig. 11. The effect of the various search transformers on the search are shown: black nodes are visited and white nodes are not. From left to right: depth bound 4, node bound 10 and limited discrepancy 1.

`Int`. Now we want a transformer that limits the overall number of wake-ups; it obviously has to be aware that the FD solver is being used to call the `wakeUps` function.

We can solve this problem by specifying what solver a transformer depends on:

```
class Solver (ForSolver t) => Transformer t where
  type ForSolver t :: * -> *
```

The above transformers of course keep on working for any solver, by adding the solver as a phantom type parameter e.g.:

```
data DepthBoundedST solver = DBST Int

instance (Solver solver)  => Transformer (DepthBoundedST solver) where
  type ForSolver (DepthBoundedST solver) = solver
```

However, we can now also express the wake-up bounded transformer, which only works for the `FD` solver:

```
newtype WakeUpBoundedST = WBST Int

instance Transformer WakeUpBoundedST where
  type EvalState WakeUpBoundedST = Int
  type TreeState WakeUpBoundedST = ()
  type ForSolver WakeUpBoundedST = FD
  initT (WBST n)        = (n,())
  nextT tree q t es ts  =
    do w <- wakeUps
       if es - w < 0 then return []
                     else eval' tree q t (es - w) ts
```

We'll see more fundamental uses of solver-dependency later.

7.3 Other Transformers

In the above, the transformer interacts with the evaluator when pushing and popping the queue. As an obvious extension of the same principle, the transformer can be made to interact at other points. We obtain a general transformer-evaluation interaction systematically, if we replace each call to `eval'` and `continue` with a call to a distinct function in the `Transformer` type class.

Randomizing Search Transformer An application of this is the randomizing search transformer, which is applied every time a `Try` node is evaluated. It randomly swaps the order of the two children of the `Try` node. The randomizing search transformer only makes use of the eval state which stores a randomly generated lazy list of Booleans indicating whether to swap the order of the next `Try` node or not.

```
newtype RandomizeST = RDST Int

instance Transformer RandomizeST where
  type EvalState RandomizeST = [Bool]
  type TreeState RandomizeST = ()
  initT (RDST seed)        = (randoms $ mkStdGen seed,())
  tryT (Try l r) q t (b:bs) ts  =
       if b then eval' (Try r l) q t bs ts
            else eval' (Try l r) q t bs ts
```

There are many other search transformers we can define in this manner including: finding only the first k solutions, adding a counter to the search (e.g. to return the number of nodes explored), changing the labelling strategies at some depth, ...

8 Composable Search Transformers

We observe that many search transformers can conceptually be composed to obtain more complex and advanced ones. For instance, a combination of limited discrepancy search and depth-bounded search, with well-chosen limits, realizes a different pruning of the tree than either independently.

Unfortunately, the approach of the previous section does not support this compositional view at all: a conceptual composition of transformers means writing a new transformer from scratch. What we want is a plug-and-play solution, where we can easily compose complex search transformers from simple ones, maximizing reuse and programmer convenience.

Composable Transformer Interface Hence, we define a new type class `CTransformer` of composable transformers:

```
class CTransformer c where
  type CEvalState c :: *
  type CTreeState c :: *
  initCT :: c -> (CEvalState c, CTreeState c)
  leftCT, rightCT :: c -> CTreeState c -> CTreeState c
  leftCT  _  = id
  rightCT    = leftCT
  nextCT :: CSearchSig solver c a
```

The interface of this type class is quite close to that of `Transformer`. The difference is hiding in the type of `nextCT`:

```
type CSearchSig solver c a =
     (Solver solver, CTransformer c)
     => Tree solver a -> c -> CEvalState c -> CTreeState c
        -> (EVAL solver c a) -> (CONTINUE solver c a) -> solver [a]
```

On the one hand, it is simpler than that of `nextT`, because the queue type is no longer mentioned: at this point we do not expect the transformer to manipulate the queue.

On the other hand, it has two new parameters: `EVAL solver c a` and `CONTINUE solver c a`. These parameters are necessary for compositionality. As the transformer now does not (necessarily) run on top of a primitive search, but (possibly) on top of a stack of other transformers, it is not allowed to call `eval'` and `continue` directly. Doing that would simply bypass all other transformers. Hence, `EVAL solver c a` is an abstraction of `eval'` that takes care of the other transformers in the stack before calling the actual `eval'`; similarly for `CONTINUE solver c a` and `continue`:

```
type EVAL     solver c a = (Tree solver a -> CEvalState c -> solver [a])
type CONTINUE solver c a = (CEvalState c -> solver [a])
```

One can think of these functions as continuations. The former proceeds with the search, while the latter aborts the search at the current location in the tree and continues with the next location in the queue.

Transformer Composition Composable transformers are composable because we can compose them into new composable transformers. In particular we compose them by stacking them.

We define the composition as a data type:

```
data Composition es ts where
  (:-) :: (CTransformer a, CTransformer b)
       => a -> b
       -> Composition (CEvalState a,CEvalState b) (CTreeState a,CTreeState b)
```

which contains two composable transformers. The components, as existential types, are hidden; only their states are exposed.

The whole point of such a composition is that it is a composable transformer again:

```
instance CTransformer (Composition es ts) where
  type CEvalState (Composition es ts) = es
  type CTreeState (Composition es ts) = ts
  initCT (c1 :- c2)   = let (es1,ts1) = initCT c1
                            (es2,ts2) = initCT c2
                        in ((es1,es2),(ts1,ts2))
  leftCT (c1 :- c2) (ts1,ts2)   = (leftCT c1 ts1,leftCT c2 ts2)
  rightCT (c1 :- c2) (ts1,ts2)  = (rightCT c1 ts1,rightCT c2 ts2)
  nextCT tree (c1 :- c2) (es1,es2) (ts1,ts2) eval' continue  =
    nextCT tree c1 es1 ts1
      (\tree' es1' -> nextCT tree' c2 es2 ts2
                      (\tree'' es2' -> eval' tree'' (es1',es2'))
```

```
                        (\es2' -> continue (es1',es2')))
 (\es1' -> continue (es1',es2))
```

The above code is fairly straightforward. The function of interest is `nextT`, which in continuation passing style first calls the first component, and then the next. Note that neither component needs to know about the other.

Composable Transformers as Transformers Now, we can turn any composable transformer (usually a stack of composable transformers) into an ordinary transformer by means of the `TStack` transformer:

```
data TStack es ts where
   TStack :: CTransformer c
          => c -> TStack (CEvalState c) (CTreeState c)

instance Transformer (TStack es ts) where
  type EvalState (TStack es ts) = es
  type TreeState (TStack es ts) = ts
  initT  (TStack c)  = initCT c
  leftT  (TStack c)  = leftCT c
  rightT (TStack c)  = rightCT c
  nextT tree q t@(TStack c) es ts =
    nextCT tree c es ts
      (\tree' es' -> eval' tree' q t es' ts)
      (\es'       -> continue q t es')
```

Here we see that the base continuations are indeed `eval'` and `continue`, as expected.

8.1 Composable Transformer Implementations

The implementation of composable transformers is much the same as that of ordinary transformers. For instance, here is the adapted, now composable, depth-bounded transformer:

```
newtype CDepthBoundedST = CDBST Int

instance CTransformer CDepthBoundedST where
  type CEvalState CDepthBoundedST = Bool
  type CTreeState CDepthBoundedST = Int
  initCT (CDBST n)  = (False,n)
  leftCT _ ts      = ts - 1
  nextCT tree c es ts eval' continue
    | ts == 0    = continue True
    | otherwise  = eval' tree es
```

Combining this one with limit 40 with a composable node-bounded transformer with limit 20, and running the search is as easy as writing:

```
solve model = run $ eval model [] (TStack (CNBST 20 :- CDBST 40))
```

Table 2. *Composable Transformer Library*

Short-Hand Notation	Description
`it`	identity transformer
`db` *n*	depth bound *n*
`nd` *n*	node bound *n*
`ld` *n*	limited discrepancy *n*
`fs`	first solution only
`ra` *n*	randomizing with seed *n*
`bb` *f*	branch-and-bound with bound updater *f*

Table 3. *Comparing the results of search transformers on default* `in order` *labelling for n queens.*

n	`it`	`ra 13`	`ld 10`	`ld 10 :- ra 13`	`ra 13 :- ld 10`
1	8	8	8	8	8
2	13	13	13	13	13
3	23	23	23	23	23
4	52	53	52	53	53
5	62	62	62	62	62
6	117	128	117	128	128
7	107	107	107	107	107
8	254	180	254	180	180
9	190	201	190	201	201
10	285	494	285	490	489
11	261	266	261	266	266
12	514	545	513	543	543
13	370	486	370	486	486
14	1,995	1,619	8,930	4,919	1,683
15	1,555	3,183	1,366	1,120	1,104
16	9,738	3,342	137,720	11,854	9,584
17	6,430	6,430	3,759	6,640	4,931

We have implemented a small library of composable transformers, summarized in Table 2. With this library, we can simply plug-and-play, and try out lots of different combinations.

For instance, we can experiment with the effect of different search strategies on finding the first solution of *n* queens. Table 3 compares randomizing search (seed 13), limited discrepancy (with limit 10) and their composition on `in order` labelling. The results show that randomizing can improve a poor labelling, while LDS can be significantly worse, and combining them ameliorates the worst of the LDS. They also illustrate how the transformers do not commute, it's better to randomise before LDS than after.

8.2 Branch-and-Bound Optimization

The branch-and-bound search strategy is a classic approach to optimization. After it has found a solution, it is only interested in finding *better* solutions next. In other words, it tries to prune the search tree by eliminating subtrees that do not yield a better solution.

The quality of a solution is based on an objective constraint variable. The value assigned to this variable determines the quality. Usually, the smaller the value is, the better the solution.

We can easily add a generic branch-and-bound strategy as a composable transformer in our framework. This transformer is parametric in a `NewBound solver` action. This action should be called when a solution is found; it returns a function, `Bound solver`, for imposing constraints on further subtrees to only look for better solutions.

```
newtype CBranchBoundST (solver :: * -> *) = CBBST (NewBound solver)

type Bound    solver  = forall a. Tree solver a -> Tree solver a
type NewBound solver  = solver (Bound solver)
```

The transformer keeps track, in its evaluation state, of the above `Bound solver` function and applies it in `nextCT`. As an optimization, we want to apply each new function only once to any given subtree. Hence, the solver keeps track of the current function's version number in its evaluation state, and of the version number of each subtree in its tree state.

```
data BBEvalState solver  = BBP Int (Bound solver)

instance Solver solver => CTransformer (CBranchBoundST solver) where
  type CEvalState (CBranchBoundST solver) = BBEvalState solver
  type CTreeState (CBranchBoundST solver) = Int
  type CForSolver (CBranchBoundST solver) = solver
  initCT _  = (BBP 0 id,0)
  nextCT tree c es@(BBP nv bound) v
    | nv > v        = evalCT (bound tree) c es nv
    | otherwise     = evalCT tree         c es v
  returnCT (CBBST newBound) (BBP v bound) continue =
    do bound' <- newBound
       continue $ BBP (v + 1) bound'
```

In the above, the `returnCT` function is an addition to the `CTransformer` type class, that allows interaction when a new solution is found. This function is used to obtain the new bounding function.

Note that `Bound solver` is a rank-2 type: it forces genericity in `a`, the result type of the whole computation.

Let us illustrate the branch-and-bound transformer for the FD solver with a small example. Assuming that `objective :: FD FDVar` returns the variable to be minimized, this is achieved by the following solver function:

```
solve model = runSM $ eval model [] (TStack (CBBST newBound))

newBound :: NewBound FD
newBound =
  do obj <- objective
     val <- value obj
     return ((\tree -> obj @< val /\ tree) :: Bound FD)
```

Whenever a new solution is found with objective value `val`, the tree transformer is changed to add a constraint demanding a *better* solution, i.e. with objective value smaller than `val`. This assumes that the variable to be minimized is fixed in any solution.

```
gmodel n  = NewVar $ \_ -> path 1 n 0

path :: Int -> Int -> Int -> Tree FD Int
path x y d = if x == y
  | x == y     =  return d
  | otherwise  = disj [ Label (fd_objective >>= \o ->
                                  return (o @> (d+d' - 1) /\
                                          (path z y (d+d'))))
                      | (z,d') <- edge x
                      ]

edge i | i < 20     = [ (i+1,4), (i+2,1) ]
       | otherwise  = []
```

Fig. 12. Code to create a search tree for finding paths.

Here is a new twist on the traditional branch-and-bound algorithm. Optimistically, we assume that a solution can be found that is twice as good, i.e. whose objective value is less than half the current solution's objective value. Hence, we look for a new solution in the bottom half of the interval $[0, \texttt{val} - 1]$ first. However, for completeness sake we also consider the upper half of that interval.

```
newBoundBis =
  do obj <- objective
     val <- value obj
     let m = val `div` 2
     return ((\tree -> (obj @< (m + 1)
                           \/
                        (obj @> m /\ obj @< val))
                        /\ tree) :: Bound FD)
```

If the optimistic assumption is right a lot of the time, we make progress more quickly than in the previous approach.

Note that this kind of optimization search, called optimistic partitioning, is commonly used in restart optimization, where we restart the search from scratch after each solution is found. But we are unaware of any literature that uses this transformation *during search*, while it's only a small change to our code.

Example 6 To illustrate branch-and-bound search we introduce a new example program. The code in Figure 12 defines a shortest path program. The solver only involves a single variable representing the path length, which is constrained by the labelling predicate `path`. `path` builds a search tree based on `edge i` which returns the list of nodes reachable from i with their edge lengths.

The graph we use is a simple linear graph with edges from i to $i+1$ of length 4, and to $i+2$ of length 1 (see Figure 13).

Table 4 compares the number of search tree nodes visited for different search strategies on the path program to find the best solution: without bounding (so looking for all

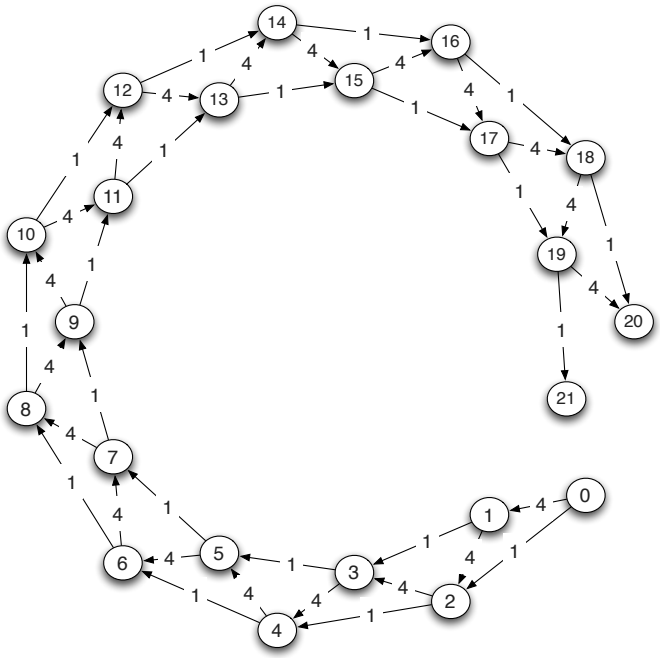

Fig. 13. Example graph for shortest path search.

solutions), using branch and bound and bisection branch and bound, and using best-first search.

Best-first search differs in the queue data type from depth-first search: a priority queue maintains the subtrees to visit. A priority is assigned based on the (heuristic) likeliness of yielding a short path. For this purpose, we use the lower-bound of the objective variable's domain, i.e. the distance so far. In other words, this best-first search realizes a greedy strategy, always extending the current shortest (partial) path.

We can see the significant improvements in using branch-and-bound search, and how bisection branch-and-bound search improves upon this. Of course for these examples best-first search, which is an *informed* search strategy is significantly better, but informed search strategies are usually not available in typical constraint programming problems. □

8.3 Restarting Transformer

Some search strategies revisit a tree—usually different parts of it—multiple times. Two typical examples are *iterative deepening* and *restart optimization*. Iterative deepening repeatedly performs a depth-bounded search, each time increasing the depth-limit. Restart optimization is similar to branch-and-bound but restarts from scratch: each time it tightens the bound on the objective variable using the previous solution found.

We propose the following generic `RestartST` transformer that captures the common pattern of iterative deepening and restart optimization. As we present this new transformer, we will introduce the necessary extensions of our framework as we go.

Firstly, the restart transfomer looks like:

Table 4. *Comparison of search strategies to find paths in the program shown in Figure 12 using: depth-first search finding all solutions (DFS ALL), depth-first search with branch and bound (DFS BB), depth-first search with bisection branch and bound (DFS BB-Bis), and best-first search (BeFS) (which requires the queueing strategy to be made aware of the distance so far).*

destination	DFS ALL	DFS BB	DFS BB-Bis	BeFS FS
1	2	2	2	2
2	33,450	45	52	5
3	20,682	56	57	9
4	25,570	110	110	13
5	23,722	176	126	17
6	24,458	252	197	21
7	24,226	395	263	25
8	24,394	504	337	29
9	24,458	757	464	33
10	24,642	910	716	37
11	24,906	1,317	780	49
12	25,354	1,525	1,147	53
13	26,050	2,141	1,220	73
14	27,242	2,426	1,777	77
15	29,002	3,048	1,734	105
16	32,290	3,433	2,487	109
17	36,458	4,265	3,086	145
18	46,218	4,793	3,425	149
19	54,114	5,978	4,266	201
20	87,562	6,704	4,718	205
21	87,562	8,416	5,883	214

```
data RestartST c a where
  RestartST :: CTransformer c
            => [c]
            -> (Tree (CForSolver c) a -> (CForSolver c) (Tree (CForSolver c) a))
            -> RestartST c a
```

I.e., it captures a sequence of composable transformers to iterate through, and a function for updating trees.

The restart transformer is a regular transformer and not a composable one:

```
instance Solver (CForSolver c) => Transformer (RestartST c a) where
  type ForSolver (RestartST c a) = CForSolver c
  type EvalState (RestartST cs a) = RestartState c a
  type TreeState (RestartST cs a) = CTreeState c
  ...
```

Most of the time, the restart transformer lets one of the captured transformers do its job. Hence, the tree state is that of c. The evaluation state is more involved:

```
data RestartState c a = RS { current    :: c
                           , next       :: [c]
                           , current_state :: CEvalState c
```

```
                                , init_label :: Label (CForSolver c)
                                , init_tree  :: Tree (CForSolver c) a }
```

i.e., it maintains the currently active composable transformer and its evaluation state, as well as the remaining composable transformers. The last two components are the initial model tree and the associated initial solver state. In order to allow for the initialization, we generalize the signature of `initT` to allow for the following implementation:

```
initT (RestartST (c:cs) _) tree  =
  let (esc,tsc) = initCT c
  in do l <-  markSM
        let es = RestartST { current       = c
                           , current_state = esc
                           , next          = cs
                           , init_label    = l
                           , init_tree     = tree }
        return (es,tsc)
```

Observe that, as the type of the evaluation state now depends on `a`, the search's result type, the transformer now also depends on that type. We express this by adding an additional associated type family to the `Transformer` class:

```
type ForResult (RestartST c a) = a
```

We omit most of the method implementations, as these are simply delegated to `current`. However, there is one new transformer method `endT` that is called when the queue has run out of elements to process. This method allows the restart transformer to start the worklist anew with the initial tree:

```
endT wl t@(RestartST _ f) es
  | null (next es)                                = return []
  | completeCT (current es) (current_state es)  = return []
  | otherwise
    = let (esc,tsc) = initCT c
      in  do tree'   <- f (init_tree es)
             let es'  = es {current = head $ next es
                           ,next    = tail $ next es}
                 node = (init_label es,tree',tsc)
             continue (pushQ node wl) t es'
```

If there are no more composable transformers, then the search should end. Otherwise, if the last run completely visited the tree, the search should end as well. We've added the `completeCT` member to the `CTransformer` type class. In all other cases, the initial tree is pushed onto the queue (after transformation by `f`), and the next composable transformer becomes the current one.

Now we can easily express iterative deepening as the restart optimization `(RestartST (map db [1..]) return)` and restart optimization as `(RestartST (repeat fs) opt)` where `opt` is defined as:

```
opt tree = do f <- newBound
              return (f tree)
```

8.4 Composable vs. Basic Transformers

As our framework provides both composable and basic search transformers, there may be a question of where to articulate a particular search strategy. Indeed, the same effect can be achieved in different ways. For instance, a combination of several composable transformers can also be implemented as a single monolithic basic search transformers.

In general, we would suggest to aim for increased flexibility. A composable search transformer is more readily reused as part of a different complex search strategy than a basic search transformer. The same holds for a basic search transformer and a primitive queue data structure versus a dedicate queue data structure with advanced behavior.

9 Related Work

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

Constraint Programming Constraint logic programming languages such as ECLiPSe (ECLiPSe, 2008) and SICStus Prolog (SICStus, 2008) 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. ECLiPSe provides a search library which allows: user programmable variable and value selection (as in Section 5.2) 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 Salsa (Laburthe & Caseau, 2002) language is an imperative domain-specific language for implementing search algorithms on top of constraint solvers. Its center of focus is a node in the search process. Programmers can write custom "Choice" strategies for generating next nodes from the current one; Salsa provides a regular-expression-like language for combining these Choices into more complex ones. In addition, Salsa allows custom procedures to be run at the *exits* of each node, i.e. right after visiting it. We believe that Salsa's Choice construct is orthogonal to our approach, and could be easily incorporated. The custom exit procedures show similarity to our transformers, but no support is provided for composing transformers.

The Oz (Smolka, 1995) language was the first language to truly separate the definition of the disjunctive constraint model from the search strategy used to explore it (Schulte, 1997). 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 original versions of the constraint programming language OPL (Van Hentenryck, 1999; Van Hentenryck *et al.*, 2000) provided a user programmable search language facility using a `try` construct, analogous to the `Dynamic` nodes in a model tree. The resulting tree could then be explored using a programmed exploration strategy (or built in exploration strategies such as DFS, LDS, BFS or BeFs). These explorations were based on a priority queue of nodes and programmed by giving a priority to each node, and well as a test to determine when to examine the next element in the queue rather than the children of the current node. This provided something equivalent to the `Queue` class. Exploration strategies could be modified by limit strategies which effectively created search transformers equivalent to our depth-bounded or node-bounded search transformers. These limit strategies appear to have been stackable.

The closest work to this paper is the search language (Van Hentenryck & Michel, 2006) of Comet (Van Hentenryck & Michel, 2005). 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 (Michel *et al.*, 2006).

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 (Somogyi *et al.*, 1996) and CIAO (Casas *et al.*, 2006). 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 (Brassel & Huch, 2007), 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 Program-

ming 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 (Claessen & Ljunglöf, 2000) and Jansson and Jeuring (Jansson & Jeuring, 1998) 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 (Wadler, 1985)—upon which various improvements have been made, e.g. breadth-first search (Seres & Spivey, 1999), Prolog's pruning operator *cut* (Hinze, 2001), and fair interleaving (Kiselyov *et al.*, 2005).

The Alma-0 (Apt *et al.*, 1998) 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 (Barnier, 2002). 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.

In recent preliminary work, Fischer (Fischer, 2008) discusses how to add constraints to any instance of `MonadPlus`, with the goal of modeling Functional Logic programming in Haskell. In his approach, the search strategy is determined by the particular `MonadPlus` instance. There are no separate provisions for a queuing type or (composable) search strategy transformers.

Search in Functional Programming Various specific instances of search-related problems have been solved in Haskell, of which the Sudoku puzzle is perhaps the most famous. While the Sudoku puzzle can be solved by many approaches, it is one at which Finite Domain constraint programming excels: state-of-the-art FD solvers solve 9-by-9 puzzles in milliseconds. Yet, of the 19 Haskell Sudoku solvers currently on `http://haskell.org/haskellwiki/Sudoku`, only one, by David Overton, considers an implementation in terms of an FD solver, and even that one implements a fixed depth-first search. Typical Haskell solutions, such as Bird's (Bird, 2006), implement problem specific solvers with a hard-wired search strategy.

10 Conclusion & 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

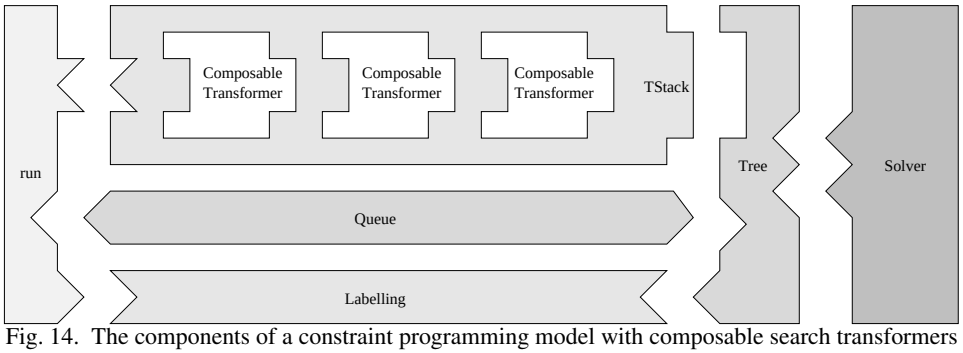

Fig. 14. The components of a constraint programming model with composable search transformers and how they fit together.

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. The components: solver, search tree, labelling, queue, and search transformers are separate but nest together as shown in Figure 14.

In future work we would like to:

- generalize our search framework to arbitrary search problems.
- integrate a Haskell implementation of Constraint Handling Rules (Frühwirth, 1998) with the framework to provide the combination of programmable **s**earch and programmable **s**olving.
- make state-of-the-art constraint solver implementations (e.g. Gecode (Schulte *et al.* , 2009)) available, by binding them to Haskell using the C foreign function interface and have them implement the `Solver` type class.
- investigate the connection between our composable search transformers and *mix-ins* (Cook, 1989), and in particular develop monadic mixins suitable for hiding the transformers' state.
- explore the performance characteristics of the framework:
  1. the overhead of the search strategy transformers with respect to the basic search strategies, and
  2. the overhead of the FFI bindings and search strategies with respect to native search strategies for state-of-the-art solvers.

Acknowledgments

We are grateful to the anonymous reviewers of this paper whose comments and suggestions have significantly improved the presentation as well as Ben Moseley, Serge Le Huitouze, Pieter Wuille and Bruno Oliveira for their helpful comments.

References

Apt, Krzysztof R., Brunekreef, Jacob, Partington, Vincent, & Schaerf, Andrea. (1998). Alma-o: an imperative language that supports declarative programming. *ACM Trans. Program. Lang. Syst.*, **20**(5), 1014–1066.

Barnier, Nicolas. 2002 (December). *Application de la programmation par contraintes à des problmes de gestion du trafic aérien*. Ph.D. thesis, Institut National Polytechnique de Toulouse. `http://www.recherche.enac.fr/opti/papers/thesis/`.

Bird, Richard S. (2006). A program to solve Sudoku. *Journal of Functional Programming*, **16**(6), 671–679.

Brassel, Bernd, & Huch, Frank. (2007). On a tighter integration of Functional and Logic Programming. *Pages 122–138 of:* Shao, Zhong (ed), *5th Asian Symposium on Programming Languages and Systems (APLAS'07)*. Lecture Notes in Computer Science, vol. 4807. Springer.

Casas, Amadeo, Cabeza, Daniel, & Hermenegildo, Manuel V. (2006). A syntactic approach to combining functional notation, lazy evaluation and higher-order in LP systems. *Pages 146–162 of: 8th International Symposium on Functional and Logic Programming (FLOPS'06)*. Springer.

Claessen, Koen, & Ljunglöf, Peter. (2000). Typed logical variables in Haskell. *Proc. of Haskell Workshop* . ACM SIGPLAN.

Cook, William R. (1989). *A denotational semantics of inheritance*. Ph.D. thesis, Brown University.

ECLiPSe. (2008). *Eclipse*. http://www.eclipse-clp.org/.

Fischer, Sebastian. (2008). *Constrained Monadic Computations*. `http://www-ps.informatik.uni-kiel.de/~sebf/projects/constraint-monad.html`.

Frühwirth, Thom. (1998). Theory and practice of Constraint Handling Rules. *J. logic programming, special issue on constraint logic programming*, **37**(1–3), 95–138.

Hinze, Ralf. (2001). Prolog's control constructs in a functional setting - axioms and implementation. *International journal of foundations of computer science*, 125–170.

Jaffar, Joxan, & Lassez, Jean-Louis. (1987). Constraint logic programming. *Pages 111–119 of: Popl*.

Jansson, Patrik, & Jeuring, Johan. (1998). Polytypic unification. *Journal of Functional Programming*, **8**(5), 527–536.

Kiselyov, Oleg, chieh Shan, Chung, Friedman, Daniel P., & Sabry, Amr. (2005). Backtracking, interleaving, and terminating monad transformers: (functional pearl). *SIGPLAN Not.*, **40**(9), 192–203.

Laburthe, François, & Caseau, Yves. (2002). SALSA: A language for search algorithms. *Constraints*, **7**(3), 255–288.

Marriott, K., & Stuckey, P.J. (1998). *Programming with constraints: an introduction*. MIT Press.

Michel, Laurent, See, Andrew, & Hentenryck, Pascal Van. (2006). High-level nondeterministic abstractions in. *Pages 359–374 of:* Benhamou, Frédéric (ed), *Cp*. Lecture Notes in Computer Science, vol. 4204. Springer.

Overton, David. (2008). *Haskell FD library*. `http://overtond.blogspot.com/2008/07/pre.html`.

Régin, Jean-Charles. (1994). A filtering algorithm for constraints of difference in csps. *Pages 362–367 of: Aaai*.

Schrijvers, T., Peyton Jones, S., Chakravarty, M., & Sulzmann., M. (2008). Type checking with open type functions. *Pages 51–62 of: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*. ACM.

Schulte, Christian. (1997). Programming constraint inference engines. *Pages 519–533 of: Principles and practice of constraint programming - cp97, proceedings*. Lecture Notes in Computer Science, vol. 1330. Springer.

Schulte, Christian. (1999). Comparing trailing and copying for constraint programming. *Pages 275–289 of:* Schreye, Danny De (ed), *Proceedings of the sixteenth international conference on logic programming*. Las Cruces, NM, USA: The MIT Press.

Schulte, Christian, *et al.* . (2009). *The Geocde generic constraint development environment*. `http://www.gecode.org/`.

Seres, Silvija, & Spivey, Michael J. 1999 (Septembr). Embedding Prolog into Haskell. *Haskell workshop'99*.

SICStus. (2008). *Sicstus prolog*. http://www.sics.se/isl/sicstuswww/site/index.html.

Smolka, Gert. (1995). The Oz programming model. *Pages 324–343 of: Computer science today*. LNCS, vol. 1000. Springer.

Somogyi, Zoltan, Henderson, Fergus, & Conway., Thomas. (1996). The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*.

Van Hentenryck, Pascal. (1999). *The OPL optimization programming language*. Cambridge, MA, USA: MIT Press.

Van Hentenryck, Pascal, & Michel, Laurent. (2005). *Constraint-based local search*. MIT Press.

Van Hentenryck, Pascal, & Michel, Laurent. (2006). Nondeterministic control for hybrid search. *Constraints*, **11**(4), 353–373.

Van Hentenryck, Pascal, Perron, Laurent, & Puget, Jean-Francois. (2000). Search and strategies in OPL. *Acm tocl*, **1**(2), 285–315.

Wadler, Philip. (1985). How to replace failure by a list of successes. *Pages 113–128 of: Proc. of a conference on Functional programming languages and computer architecture*. New York, NY, USA: Springer-Verlag New York, Inc.