Modular Lazy Search for Constraint Satisfaction Problems∗

THOMAS NORDIN ANDREW TOLMACH

Pacific Software Research Center
Oregon Graduate Institute & Portland State University
(*e-mail:* nordin@cse.ogi.edu) (*e-mail:* apt@cs.pdx.edu)

Abstract

We describe a unified, lazy, declarative framework for solving constraint satisfaction problems, an important subclass of combinatorial search problems. These problems are both practically significant and computationally hard. Finding solutions involves combining good general-purpose search algorithms with problem-specific heuristics. Conventional imperative algorithms are usually implemented and presented monolithically, which makes them hard to understand and reuse, even though new algorithms often are combinations of simpler ones. Lazy functional languages, such as Haskell, encourage modular structuring of search algorithms by separating the generation and testing of potential solutions into distinct functions communicating through an explicit, lazy intermediate data structure. But only relatively simple search algorithms have been treated this way in the past.

Our framework uses a generic generation and pruning algorithm parameterized by a labeling function that annotates search trees with conflict sets. We show that many advanced imperative search algorithms, including conflict-directed backjumping, backmarking, minimal forward checking, and fail-first dynamic variable ordering, can be obtained by suitable instantiation of the labeling function. More importantly, arbitrary combinations of these algorithms can be built by simply composing their labeling functions. Our modular algorithms are as efficient as the monolithic imperative algorithms in the sense that they make the same number of consistency checks, and most of our algorithms are within a constant factor of their imperative counterparts in runtime and space usage. We believe our framework is especially well-suited for experimenting to find good combinations of algorithms for specific problems.

1 Introduction

Combinatorial search problems offer a great challenge to the academic researcher: they are of tremendous interest to commercial users and they are often very computationally intensive to solve. Over the past several decades the AI community has responded to this challenge by producing a steady stream of improvements to generic search algorithms. There have also been numerous attempts to organize the various algorithms into standardized frameworks for comparison (e.g., (Tsang, 1993; Kondrak, 1994; Frost, 1997)).

∗ Work supported, in part, by the US Air Force Materiel Command under contract F19628-96-C-0161, and by NSF grant CA-9703218.

Although the speed and cunning of search algorithms have improved, the new algorithms are more complicated and harder to understand, even though they are often combinations of simpler, standard algorithms. The problem is exacerbated by the fact that most algorithms are described by large, monolithic chunks of pseudocode or C code. Although it is recognized that most problems benefit from a tailor-made solution involving a combination of existing generic and domain-specific algorithms, modularity has not been a strong point of most recent research. It is difficult to reuse code except via cut-and-paste. Moreover, to prove these algorithms correct we must resort to complex reasoning about their dynamic behavior. For example, although most of these search algorithms are conceived as varieties of "tree search," no actual tree data structures appear in their implementations; trees are present only virtually, in the form of recursive routine activation histories. Perhaps for this reason, even widely-used and well-studied algorithms often lack correctness proofs.

In the world of lazy functional programming, the idea of implementing search algorithms using modular techniques is a commonplace. The classic paper of Hughes (1989) and the textbook of Bird & Wadler (1988) both give examples of search algorithms in which generation and testing of candidate solutions are separated into distinct phases, glued together using an explicit, lazy, intermediate data structure. This "generate-and-test" paradigm makes essential use of laziness to synchronize the two functions (really coroutines) in such a way that we never need to store much of the (exponential-sized) intermediate data structure at any one time. In general, the modular lazy approach can lead to algorithms that are much simpler to read, write, and modify than their imperative counterparts. However, the algorithms described in these sources are fairly elementary.

In this paper we present a lazy declarative framework for solving one important class of combinatorial search problems, namely constraint satisfaction problems (CSPs) over finite domains. This class of problems includes graph coloring and matching, scene labeling in computer vision, temporal reasoning, resource allocation in planning and scheduling, and many others (Tsang, 1993). For simplicity, we restrict our attention to binary CSPs, but this restriction is not fundamental to our general approach. Our framework is based on explicit, lazy, tree structures, in which each tree node represents a *state* in the search space; problem solutions correspond to leaf nodes that meet certain criteria. Nodes can be labeled with *conflict sets*, which record constraint violations in the corresponding states; many algorithms use these sets to *prune* subtrees that cannot contribute a solution.

Our code is written in Haskell 98 (Peyton Jones & Hughes, 1999). We provide a small library of separate functions for generating, labeling, rearranging, pruning, and collecting solutions from trees. In particular, we describe a generic search algorithm, parameterized by a labeling function, and show that a variety of standard imperative CSP algorithms, including simple backtracking, conflict-directed backjumping (Prosser, 1993a), backmarking (Gaschnig, 1977), and minimal forward checking (Dent & Mercer, 1994), can be obtained by making a suitable choice of labeling function. A further class of algorithms based on dynamic variable ordering (Kumar, 1992) can be obtained via a small change to the generating function. Using an explicit representation of the search tree allows us to focus on the data associated with each search state and gives us new insights into more efficient algorithms. As in other recent work on functional algorithms and data structures (King & Launchbury, 1995; Okasaki, 1998), we found that recasting imperative algorithms into a declarative lazy setting casts new light on the fundamental algorithmic ideas. In particular, it is easy to see how to combine our algorithms, simply by composing their labeling functions, and to see that the result will be correct.

Since the whole point of improving search algorithms is to be able to solve larger problems faster, we must obviously be concerned with the performance of our lazy algorithms. Our experiments show that lazy, modular Haskell code is an order of magnitude slower than a direct recursive implementation in Haskell; moreover, even the latter can be several times slower than the equivalent C code. However, since search times often explode exponentially, even slowdowns of one or two orders of magnitude have little effect on the size of problem we can solve within a fixed time bound. All our algorithms and their combinations are fast enough for experiments that have been interesting to researchers in the past; for example we are able to reproduce parts of the comparative tables assembled by Bacchus and van Run (1995) and Kondrak (1994). More importantly, our implementations are fast enough to allow experimentation with different combinations of algorithms on problems of realistic size. For such experiments, CPU time is often not an ideal comparison metric, since it is difficult to compare times obtained from different implementations on different systems. A widely used alternative metric is the number of consistency checks performed by the algorithm, and we adopt this metric here.

Although the generate-and-test algorithms we discuss in this paper are widely applicable, they are too low-level to take advantage of the specific characteristics of many real-world problems, which are often better handled by explicit constraint manipulation. For example, our framework could be used directly to solve scheduling problems over discrete ordered domains by brute-force search, but it would typically be much more efficient to represent scheduling constraints using intervals and to apply interval-based reasoning to reduce the domains of variables before resorting to generate-and-test. Thus, a more comprehensive system for solving constraint problems might include our framework as just one component among several.

The paper is organized as follows. Section 2 formalizes our problem domain and Section 3 gives a Haskell specification for it. Section 4 describes simple tree-based backtracking search. Section 5 introduces conflict sets and our generic search algorithm, and recasts backtracking search in that framework. Section 6 briefly discusses search heuristics based on value reordering. Section 7 describes how the conflict set framework can be used to support more intelligent backtracking. Sections 8 and 9 describe more sophisticated algorithms based on the idea of caching consistency checks, and Section 10 discusses how algorithms can be combined. Section 11 extends these ideas to dynamic variable ordering. Section 12 summarizes performance results, Section 13 describes related work, and Section 14 concludes.

The reader is assumed to have a working knowledge of lazy functional programming and a reading knowledge of Haskell, although certain Haskell subtleties will be explained as they arise. All the code examples in this paper and additional required support code are available from the journal website.

2 Binary Constraint Satisfaction Problems

Definition 1
A *binary constraint satisfaction problem* is described by

- a set of variables $V = \{v_1, v_2, \ldots, v_m\}$;
- for each variable v_i, a finite set D_i of possible values (its *domain*); and
- for each pair of distinct variables v_i and v_j, $i < j$, a binary relation $R_{ij} \subseteq D_i \times D_j$, representing a *constraint* on the values that v_i and v_j can take on simultaneously.

An *assignment* v_i`:=`x_i associates a variable v_i to some value $x_i \in D_i$. A *state* is a set of assignments, with at most one assignment per variable. A state S' *extends* state S if it contains all the assignments of S together with one or more additional assignments.

A pair of assignments v_i`:=`x_i and v_j`:=`x_j, $i < j$, *satisfies* the corresponding constraint R_{ij} if $(x_i, x_j) \in R_{ij}$. A state is *consistent* if every pair of distinct assignments in the state satisfies the corresponding constraint; otherwise it is *inconsistent*.

A state is *complete* if it assigns all the variables of V; otherwise it is *partial*. A *solution* to a CSP is any complete consistent state.

This definition of binary CSPs can be generalized by replacing the binary relations by n-ary relations. Although our general approach should extend to this broader class of problems, the algorithms in this paper rely heavily on the binary nature of the constraints. In any case, an n-ary CSP can always be encoded (though not necessarily efficiently) as an equivalent binary CSP (Stergiou & Walsh, 1999).

To simplify the presentation in this paper, we assume that all domains have the same size n and that their values are represented by integers in the set $\{1, 2, \ldots, n\}$; these limitations could be trivially removed.

A naive approach to solving a CSP is to enumerate all possible complete states and then check each in turn for consistency. In a binary CSP, consistency of a state can be determined by performing *consistency checks* on each pair of assignments in the state, until an *inconsistent pair* of variables is detected, or all pairs have been checked. Following the conventions of the search literature, we use the number of consistency checks as a key measure of algorithm efficiency, although it is not necessarily an accurate predictor of execution time.

For some problems we want to calculate all solutions, but for others we only wish to find one solution as quickly as possible. All the search algorithms in this paper are suited to either situation; the heuristics in Section 6 are specifically designed to speed up the search for a first solution.

3 CSPs in Haskell

Figure 1 gives a Haskell framework for describing CSP problems. An assignment is constructed using the infix constructor `:=`. Variables and values are numbered beginning from 1. A CSP is modeled as a Haskell record containing the number of variables, `vars`, the size of their domain, `vals`, and a constraint relation, `rel`; many

```
type Var = Int
type Value = Int

data Assignment = Var := Value

var :: Assignment -> Var
var (var := _) = var

value :: Assignment -> Value
value (_ := val) = val

type Relation = Assignment -> Assignment -> Bool

data CSP = CSP {vars, vals :: Int, rel :: Relation}

data State = State ([Assignment],[Var])

assignments :: State -> [Assignment]
assignments (State(as,_)) = as

unassigned :: State -> [Var]
unassigned (State(_,us)) = us

emptyState :: CSP -> State
emptyState CSP{vars=vars} = State([],[1..vars])

isEmptyState :: State -> Bool
isEmptyState = null . assignments

extensions :: CSP -> State -> [State]
extensions CSP{vals=vals} (State(as,nextvar:rest)) =
       [State((nextvar := val):as,rest) | val <- [1..vals]]
extensions _              (State(_,[])) = []

newNextVar :: State -> Var -> State
newNextVar s@(State(as,[])) _    = s
newNextVar   (State(as,us)) next = State(as,next:delete next us)

complete :: State -> Bool
complete = null . unassigned

lastAssignment :: State -> Assignment
lastAssignment = head . assignments

nextVar :: State -> Var
nextVar = head . unassigned
```

Fig. 1. A formulation of CSPs in Haskell.

of our functions take this record as a parameter.[1] We represent the relation as an oracle function taking two assignments and returning `True` iff the assignments obey the relevant constraint. For convenience, we require that the oracle function be symmetric (i.e., $\forall a, b.$`rel` a $b =$ `rel` b a), so that its two arguments can be passed in

[1] Functions often reference only some of these parameters; in Haskell, it is possible to pattern match against a subset of the fields of a record, as in, for example, the `emptyState` function. This function also illustrates that the same identifier (here `vars`) can be used both as a field name and as the corresponding pattern variable name.

```
generate :: CSP -> [State]
generate csp@CSP{vars=vars} = g vars
  where g 0 = [emptyState csp]
        g var = concat [extensions csp st | st <- g (var-1)]

inconsistencies :: CSP -> State -> [(Var, Var)]
inconsistencies CSP{rel=rel} st =
  [ (var a, var b) | a <- as, b <- as, var a > var b, not (rel a b) ]
        where as = assignments st

consistent :: CSP -> State -> Bool
consistent csp = null . (inconsistencies csp)

test :: CSP -> [State] -> [State]
test csp = filter (consistent csp)

solver :: CSP -> [State]
solver csp  = test csp candidates where candidates = generate csp
```

Fig. 2. A naive solver for CSPs.

either order. Using an oracle function permits great flexibility in the representation of constraints; for example, they can be implemented by a four-dimensional array of booleans or by a mathematical formula. However, the "black box" character of the oracle does prevent the use of algorithms that analyze constraint structure, such as arc consistency maintenance (Kumar, 1992); changing our code to use a less abstract constraint representation would be straightforward.

A state is modeled as a sequence of assignments, together with a sequence of as yet unassigned variables. States are built from `emptyState` by repeated use of `extensions`, which takes a state, extracts the head (if any) of its list of unassigned variables, constructs assignments of this variable to each possible value, extends the original state with each of these assignments in turn, and returns the resulting list of extended states. The order of unassigned variables in each state thus governs the order of assignments in its extensions. Ordinarily, the unassigned variables are simply listed in increasing numeric order, as set by `emptyState`; however, the head of the unassigned list can be changed using `newNextVar` (which we use only in Section 11). The `lastAssignment` operator returns the assignment with which the state was most recently extended.

Figure 2 shows an implementation of a naive solver. We present the solver in the standard "lazy pipeline" style that separates generation of candidate solutions (here the set of all complete states) from consistency testing. Although this code appears to produce a huge intermediate list data structure `candidates`, lazy evaluation insures that list elements are generated only on demand, and elements that fail the filter in `test` can be garbage collected immediately. Similarly, although `inconsistencies` appears to build a list of *all* inconsistent variable pairs in the state, `consistent` actually demands just enough of the list to check whether it is `null`, and hence at most one inconsistent variable pair is calculated. Finally, although the solver returns a list of all solutions if demanded, it can be used to obtain just the first solution (and do no further computation) by asking for just

```
queens :: Int -> CSP
queens n = CSP{vals=n,vars=n,rel=safe}
  where safe (col1 := row1) (col2 := row2) =
         (row1 /= row2) && abs (col1 - col2) /= abs (row1 - row2)

graphcoloring :: Int -> ((Var,Var) -> Bool) -> Int -> CSP
graphcoloring nodes adj colors = CSP{vars=nodes,vals=colors,rel=ok}
   where ok (n1 := c1) (n2 := c2) = c1 /= c2 || not (adj (n1,n2))
```

Fig. 3. CSP examples.

the head of the result. Although the code thus uses much less space than a strict reading would suggest, this solver is still extremely inefficient because it duplicates work, but it serves to illustrate lazy coding style and as a specification for the more sophisticated solvers we introduce beginning in Section 4.

Figure 3 shows two simple examples of CSPs that are useful for illustrating different search strategies. The n-queens problem looks for a way to put n queens on a $n \times n$ chess board such that no queen is threatening another. Our definition of `queens` is parameterized by the board size and uses the standard optimization that we only try to place one queen in each column (Nadel, 1990). The CSP variables are the columns, the values are the rows, and each assignment represents the placement of a single queen; the oracle function replies `True` on a pair of queen positions provided that the queens are on different rows and on different diagonals. We make heavy use of this example in the remainder of the paper.

The `graphcoloring` function constructs an instance of a graph coloring problem (Kempe, 1879), specified by a number of nodes, a set of edges between nodes (represented by a characteristic function on pairs of nodes), and a number of colors. The CSP variables are the graph nodes, the values are the possible colors, and the oracle function returns `True` on a pair of color assignments provided that the colors are different or there is no edge between the nodes.

Given the definition of a CSP, we can apply the general-purpose CSP machinery to solve it; for example, the expression `solver (queens 5)` generates a list of solutions to the 5-queens problem.

4 Backtracking and Tree Search

The most obvious defect of the naive solver is that it can duplicate a tremendous amount of work by repeatedly checking the consistency of assignments that are common to many complete states. A fundamental fact about CSPs is that no extension to an inconsistent state can ever be consistent, so there is no point in searching such extensions for a solution. This observation immediately suggests a better solver algorithm. A *backtracking* solver searches for solutions by constructing and checking *partial* states, beginning with the empty state and extending with one assignment at a time. Whenever the solver discovers an inconsistent state, it immediately *backtracks* to try a different assignment, thus avoiding the fruitless exploration of that state's extensions. Moreover, consistency of each new state can

```
data Tree a = Node a [Tree a]

mkTree :: a -> [Tree a] -> Tree a
mkTree a ts = Node a ts

label :: Tree a -> a
label (Node a _) = a

initTree :: (a -> [a]) -> a -> Tree a
initTree f a = Node a (map (initTree f) (f a))

mapTree  :: (a -> b) -> Tree a -> Tree b
mapTree f (Node a ts) = Node (f a) (map (mapTree f) ts)

foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree f (Node a ts) = f a (map (foldTree f) ts)

zipTreesWith :: (a -> b -> c) -> Tree a -> Tree b -> Tree c
zipTreesWith f (Node a ts) (Node b us) =
                  Node (f a b) (zipWith (zipTreesWith f) ts us)

prune :: (a -> Bool) -> Tree a -> Tree a
prune p = foldTree f
  where f a ts = Node a (filter (not . p . label) ts)

leaves :: Tree a -> [a]
leaves = foldTree f
   where f leaf [] = [leaf]
         f _    ts = concat ts

inhTree :: (b -> a -> b) -> b -> Tree a -> Tree b
inhTree f b (Node a ts) = Node b' (map (inhTree f b') ts)
    where b' = f b a

distrTree :: (a -> [b]) -> b -> Tree a -> Tree b
distrTree f b (Node a ts) = Node b (zipWith (distrTree f) (f a) ts)
```

Fig. 4. Trees in Haskell.

be tested just by comparing the newly added assignment to all previous assignments in the state, since any inconsistency involving *only* the previous assignments would have been discovered earlier. If the solver manages to reach a complete state without encountering an inconsistency, it records a solution; if multiple solutions are wanted, it backtracks to find the others.

Backtracking solvers can be viewed very naturally as searching a *tree*, in which each node corresponds to a state and the descendents of a node correspond to extensions of its state. In conventional imperative implementations of backtracking, the tree is not explicit in the program; if a recursive implementation is used, the tree is isomorphic to the dynamic activation history tree of the program, but usually the tree is little more than a metaphor for helping the programmer reason informally about the algorithm. In the lazy functional paradigm it is natural to treat search trees as *explicit* data structures; programs are constructed as pipelines of operations that build, label, manipulate, and prune actual trees. As before, we rely on laziness to avoid actually building the entire tree.

```
mkSearchTree :: CSP -> Tree State
mkSearchTree csp = initTree (extensions csp) (emptyState csp)

earliestInconsistency:: CSP -> State -> Maybe Var
earliestInconsistency CSP{rel=rel} st =
  case assignments st of
    [] -> Nothing
    (a:as) -> case filter (not . rel a) (reverse as) of
                [] -> Nothing
                (b:_) -> Just(var b)

labelInconsistencies :: CSP -> Tree State -> Tree (State,Maybe Var)
labelInconsistencies csp = mapTree f
    where f s = (s,earliestInconsistency csp s)

btsolver0 :: CSP -> [State]
btsolver0 csp =
  (filter complete . map fst . leaves . prune ((/= Nothing) . snd)
                             . (labelInconsistencies csp) .  mkSearchTree) csp
```

Fig. 5. Simple backtracking solver for CSPs.

As the remainder of this paper deals exclusively with tree-based searches, it will prove convenient to blur the distinction between a node and its associated state. Thus we will freely use terms such as *inconsistent node* (meaning a node whose associated state is inconsistent) and the *children of a state* (meaning its extensions by a single assignment).

Figure 4 gives Haskell definitions for an abstract tree datatype and associated utility functions. A `Tree` is a node containing a label and a list of children, themselves `Tree`s. Function `initTree` generates a tree from a function that computes the children of a node (Hughes, 1989). Functions `mapTree`, `foldTree`, and `zipTreesWith` are the analogues of the familiar functions on lists. The application (`prune` p t) removes all subtrees of t whose root labels match p. However, the root node of the overall tree is always retained; in our applications this is always appropriate, and it avoids the awkward possibility of an empty result, which is not expressible as a `Tree`. The `leaves` operator extracts the labels of the leaves of a tree into a list in left-to-right order. The `inhTree` function is a variant of map that propagates a value down the tree much like an inherited attribute calculation in an attribute grammar, and `distrTree` is another map variant that transforms the value at each node to a list whose elements are distributed to the children.

The code in Figure 5 uses trees of states to implement a backtracking solver, `btsolver0`, using a lazy pipeline. The generator `mkSearchTree` builds a tree containing all possible states, using a fixed variable ordering in which all nodes at level i of the tree (counting the root as level 0) extend their parent by an assignment to a fixed v_i. Each node describes an entire (partial) state, but sharing of list tails (in any reasonable Haskell implementation) guarantees that it actually stores only a single assignment, together with a pointer to the remainder of the state embedded in its parent node.

The application (`labelInconsistencies` *csp*) returns a *tree transformer*: it

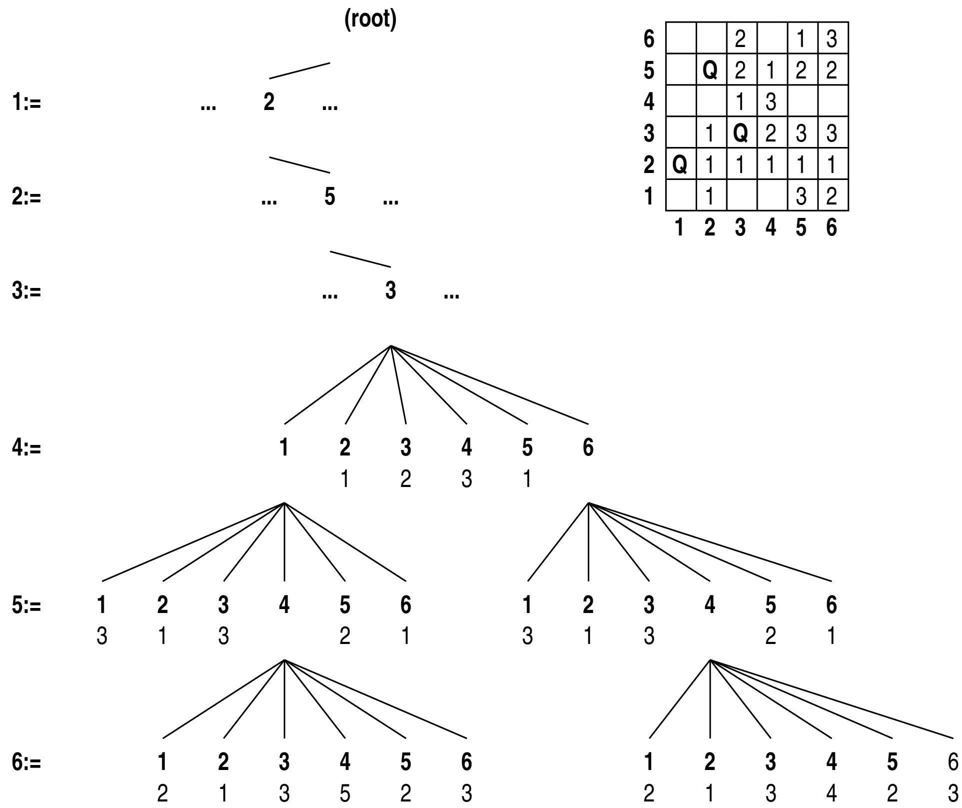

Fig. 6. Portion of search tree for `queens 6`. Nodes at level i show assignments to v_i; the assigned values are shown in **bold**; the earliest variable, if any, with which the node is inconsistent is shown below it. Children of inconsistent nodes are not shown because, under lazy evaluation, they are never constructed. The diagram at the top right shows the board position corresponding to the node at level 3. Each placed queen is marked by a Q in the corresponding square. If a square is numbered, then it is threatened by a placed queen; the number is the column of the left-most queen that threatens that square.

adds an annotation to each node recording the index of an earlier variable with which the most recent assignment conflicts, if any. In fact, `earliestInconsistency` returns the *earliest* such variable (i.e. the one least recently assigned); the point of choosing this variable will become apparent in Section 7. The tree function `prune` is used to remove all subtrees rooted at inconsistent nodes. Any nodes representing complete states that are still left in the tree must be solutions; the remaining pipeline stages extract these using the tree function `leaves` and the standard list functions `map` and `filter`. Figure 6, taken from Kondrak (1994), illustrates a part of the tree for `queens 6` just prior to pruning. It is essential to note that this pipeline is demand driven: each stage executes only when demanded by the following stage. In particular, inconsistency calculations will *not* be performed on descendents of the nodes of the tree excised by `prune`, because the values of these nodes will never be demanded. Thus we get the desired effect of backtracking without any explicit manipulation of control flow. Also, as before, only a small part of each intermediate tree is ever "live" (non-garbage data) when a particular node is being operated upon, namely the node's own label and thunks both for its descendents and for right siblings of its ancestors—essentially what would be stored in activation records for a recursive imperative implementation. (In particular, the ancestors of the node and their left siblings do *not* remain live.) So our lazy algorithms pay at worst a constant factor more space than their imperative counterparts. We do, however, pay some overhead for building, storing, and garbage collecting each tree node, and, unless our Haskell implementation performs effective deforestation (Gill *et al.*, 1993), this cost will be repeated for each intermediate tree in the pipeline. For these reasons, the modular, lazy implementation of backtracking is an order of magnitude slower than a conventional recursive implementation in Haskell (see Section 12).

5 Conflict Sets and Generic Search

The utility of the backtracking solver is based on its ability to prune subtrees rooted at inconsistent nodes; it does nothing with consistent nodes. Of course, just because a state is consistent does not mean it can be extended to a solution; the assignments already made may be inconsistent with any possible choices for future variables. Figure 6 shows several examples; for instance, the state with last assignment `4 := 1` is consistent, but cannot be extended to a solution.

If a solver could identify such *conflicted* states, it could prune their extensions too. To make precise the exact conditions under which such pruning is possible, we introduce the notion of conflict set.

Definition 2
Let $s = \{v_{i_k}\texttt{:=}y_k, \ldots, v_{i_1}\texttt{:=}y_1\}$ be a state. A *conflict set CS* for s is a non-empty subset of $\{i_1, i_2, \ldots, i_k\}$ such that, for any solution $\{v_{i_m}\texttt{:=}x_m, \ldots, v_{i_1}\texttt{:=}x_1\}$, $\exists i \in CS$ such that $y_i \neq x_i$.

In other words, a conflict set for a state identifies a subset of assignments in the state such that *any* solution must assign a *different* value to at least one member of the subset. (Thinking imperatively, we might say a conflict set contains variables at least one of which "must be changed" to reach a solution.) Although use of conflict sets is very common in the literature, a precise definition is difficult to achieve; we base ours on that of Caldwell *et al.* (1997). If a conflict set exists for a state, then evidently no extension of that state can be a solution. Note that a conflict set for a given state is not, in general, uniquely defined. In particular, if a state $s = \{v_{i_k}\texttt{:=}x_k, \ldots, v_{i_1}\texttt{:=}x_1\}$ has a conflict set CS, then every subset of $\{i_1, \ldots, i_k\}$ that is a superset of CS is also a conflict set for s.

It is obviously not possible to identify a conflicted, but consistent, state without exploring *some* of its extensions; the trick is to avoid exploring all of them, and save effort by pruning the remainder. We address algorithms with this property beginning in Section 7. For the moment, note that any inconsistent state has a

```
type ConflictSet = OrderedSet Var

isConflict :: ConflictSet -> Bool
isConflict = not . isEmptySet

solutions :: Tree (State, ConflictSet) -> [State]
solutions = filter complete . map fst . leaves . prune (isConflict . snd)

type Labeler = CSP -> Tree State -> Tree (State, ConflictSet)

search :: Labeler -> CSP -> [State]
search labeler csp = (solutions . (labeler csp) . mkSearchTree) csp

bt :: Labeler
bt csp = mapTree f
  where f s = (s, case earliestInconsistency csp s of
                    Nothing -> emptySet
                    Just a -> listToSet [var (lastAssignment s),a])

btsolver :: CSP -> [State]
btsolver = search bt
```

Fig. 7. Conflict-directed solving of CSPs.

```
emptySet :: Ord a => OrderedSet a
isEmptySet :: Ord a => OrderedSet a -> Bool
memberSet :: Ord a => OrderedSet a -> a -> Bool
unionSet :: Ord a => OrderedSet a -> OrderedSet a -> OrderedSet a
intersectSet :: Ord a => OrderedSet a -> OrderedSet a -> OrderedSet a
removeFromSet :: Ord a => a -> OrderedSet a -> OrderedSet a
listToSet :: Ord a => [a] -> OrderedSet a
evalSet :: Ord a => OrderedSet a -> OrderedSet a
```

Fig. 8. Signature for ordered set ADT.

conflict set. In particular, if a state whose last assigned variable is v_i has an earliest inconsistent variable v_j, then it has $\{i, j\}$ as a conflict set, which we call the *earliest conflict set*.

A *conflict set labeling* is a state tree in which each node has been annotated with a (non-empty) conflict set for that node's state, or with the empty set, signifying that the conflict set for that node's state is unknown. To be useful for search, a conflict set labeling must reflect basic consistency information.

Definition 3
A conflict set labeling is a *searchable labeling* if, for every inconsistent node s, s or some ancestor of s is labeled with a (non-empty) conflict set.

One searchable labeling is given by annotating each node with its earliest conflict set if it has one, and with the empty set otherwise.

Using searchable labelings, we can subsume backtracking search in a more general algorithm we call conflict-directed search, shown in Figure 7. We define a generic routine `search`, parameterized by a `labeler` function that generates searchable labelings. By applying `search` to the labeler function `bt` we obtain a simple backtracking solver `btsolver` that behaves just like `btsolver0`. All the more sophisticated search algorithms discussed in the remainder of the paper, except those of Section 11, can be obtained by using fancier labeler functions while leaving `search` itself unchanged.

Conflict sets are represented using an abstract data type of ordered sets, with the signature shown in Figure 8 and the usual semantics. Our implementation (not shown here) uses ordered lists; it turns out that common operations on conflict sets are most efficient if the lists are held in decreasing numeric order. Function `evalSet` forces strict evaluation of a set's contents; it can be implemented by forcing evaluation of the representation list's length.

The structure of `search` is straightforward. The full tree of possible states is generated and fed to the labeler. If the labeler can determine a conflict set CS for a node, it annotates the node with CS; otherwise, it annotates it with $\emptyset$. (In general, we also permit the labeler to rearrange or prune its input tree, so long as its output is a searchable labeling and still contains all solution states.) The output of the labeling stage is fed to a pruner, which removes subtrees rooted at nodes labeled with conflict sets. Again, demand-driven execution guarantees that the excised subtrees never need to be labeled. This is why a searchable labeling need not attach conflict sets to the descendents of inconsistent nodes, which allows simpler and less expensive labeler code. After pruning, the solution nodes are just the complete leaves of the remaining tree; the remainder of the pipeline simply collects these. Figure 9 shows the conflict set labeling for the same `(queens 6)` subtree as in Figure 6.

6 Heuristics and Search Order

As with the naive solver, if we are interested in only the first solution rather than all solutions, we can still use `search` unchanged, by demanding just the head of the solution list. Since solutions are always extracted in left-to-right order, this implies that the time required to find the first solution will be very sensitive to the order in which values are tried for each variable. The use of *value-ordering* heuristics is well-established in the imperative search literature (Kumar, 1992). Such heuristics could be implemented using specialized generator functions that produce the initial tree in the desired order. A more modular approach, however, is to view these heuristics as ways to *rearrange* existing trees; this keeps the initial generator simple and allows multiple heuristics to be readily composed.

Such rearrangement heuristics can be easily expressed in our framework by incorporating them into the `labeler` function. For example, in solutions to the n-queens problem, queens seldom end up near the corners of the board. Therefore, `queens` search can be speeded up by considering row values in random order rather than in the usual generated order, which tends to consider corner positions first. Function `hrandom` in Figure 10 transforms a tree by randomizing the order of its children (using a random number generator not shown here). The randomization transfor-

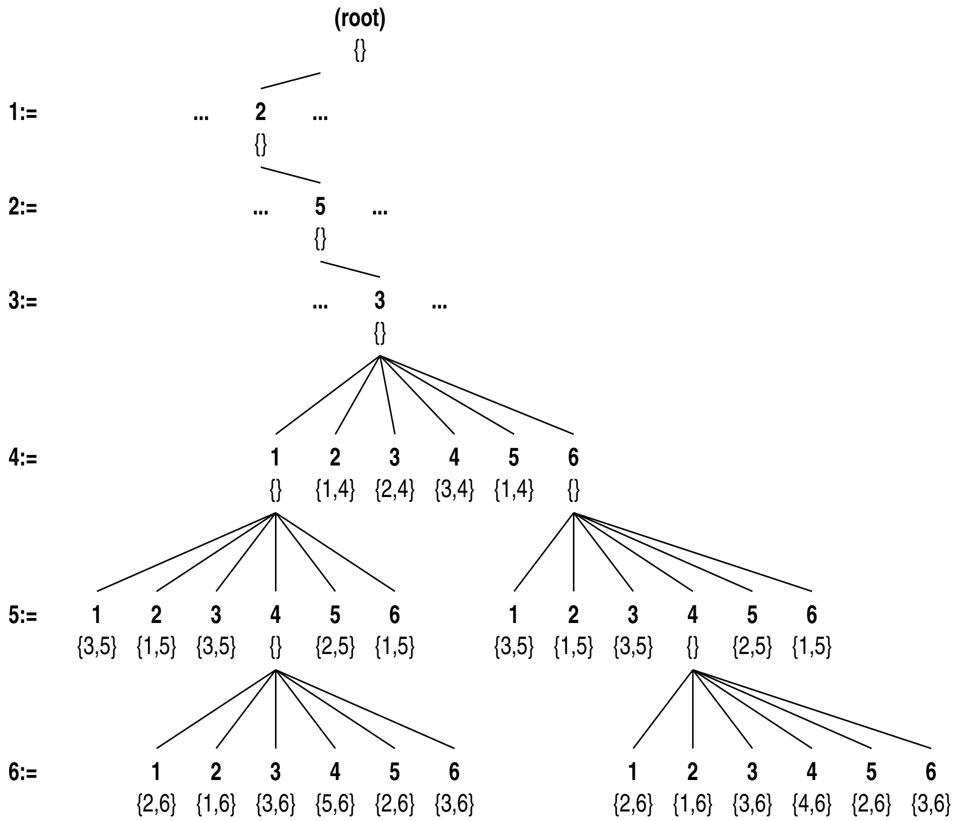

Fig. 9. Portion of search tree for `queens 6` annotated with earliest conflict sets. Nodes at level i show assignments to v_i; the assigned value x_i is shown in **bold**. Children of nodes with conflict sets are not shown because, under lazy evaluation, they are never constructed.

mation is expressed as a higher-order fold.[2] The application (`btr` *seed*) returns a labeler that combines randomization with standard backtracking search.

7 Conflict-Directed Backjumping

The `bt` algorithm annotates inconsistent nodes with conflict sets, but most internal nodes remain unannotated. If we could somehow compute conflict sets for internal nodes closer to the root of the tree, we could prune larger subtrees and so speed up search.

One approach to computing internal node conflict sets is to construct them bottom-up from the conflict sets of a subset of their children. To do this, we make use of two key lemmas about conflict sets. Intuitively, the first lemma says that if no

[2] The operator `($):(a->b)->a->b` represents explicit application in Haskell, i.e. `($) f x = f x`.

```
hrandom :: Int -> Tree a -> Tree a

hrandom seed t = foldTree g t seed
   where g a ts seed =
              mkTree a (randomizeList seed' (zipWith ($) ts (randoms seed')))
                where seed' = random seed

btr :: Int -> Labeler
btr seed csp = bt csp . hrandom seed
```

Fig. 10. A randomization heuristic.

child of a node s can be extended to a solution, then neither can s, and any solution must differ from s on the value of at least one of the variables in the conflict set of one of the children. The second lemma says that if a node s has conflicts that do not depend on the value of the last assignment in s, then the same conflicts must apply to its parent. To avoid cumbersome notation, we state and prove the lemmas assuming a fixed variable order $v_1, \ldots, v_m$, but analogous results hold for the dynamically ordered trees of Section 11.

Lemma 1
Consider a fixed-variable-order search tree for a CSP with m variables and n values. Let

$$s = (v_l := y_l, \ldots, v_1 := y_1)$$

be a node at level l $(1 \leq l < m)$ with children $s_1, \ldots, s_n$, such that each child s_i has a (non-empty) conflict set CS_i. Then

$$CS = (CS_1 \cup CS_2 \cup \cdots \cup CS_n) - \{l+1\}$$

is a conflict set for s.

Proof
Without loss of generality, assume that the children are ordered so that s_i assigns the value i to v_{l+1}, that is,

$$s_i = (v_{l+1} := y_{l+1} = i, v_l := y_l, \ldots, v_1 := y_1)\ (1 \leq i \leq n)$$

Now consider any CSP solution

$$(v_m := x_m, \ldots, v_{l+1} := x_{l+1} = k, \ldots, v_1 := x_1)$$

where we write k for x_{l+1}. Consider the conflict set CS_k associated with child s_k. By the definition of conflict set, $\exists i \in CS_k \subseteq \{1, \ldots, l+1\}$ such that $y_i \neq x_i$. However, since s_k assigns k to v_{l+1}, we cannot have $i = l+1$. Hence we must have $i \in CS_k - \{l+1\}$. Hence $i \in CS$, so CS is indeed a conflict set for s. □

Lemma 2
Consider a fixed-variable-order search tree for a CSP with m variables. Let

$$s = (v_l := y_l, \ldots, v_1 := y_1)$$

```
bj0bt :: Labeler
bj0bt csp = bj0 csp . bt csp

bj0 :: CSP -> Tree (State, ConflictSet) -> Tree (State, ConflictSet)
bj0 csp = foldTree f
  where f (s, cs) ts
          | isConflict cs  = mkTree (s, cs)  ts
          | otherwise      = mkTree (s, cs') ts
              where cs' = combine (map label ts) []

unionCS :: [ConflictSet] -> ConflictSet
unionCS css = foldr unionSet emptySet css

combine :: [(State, ConflictSet)] -> [ConflictSet] -> ConflictSet
combine []           acc = unionCS acc
combine ((s, cs):ns) acc
  | not (memberSet cs lastvar) = cs
  | isEmptySet cs = emptySet
  | otherwise = combine ns ((removeFromSet lastvar cs):acc)
      where lastvar = var (lastAssignment s)

bjbt :: Labeler
bjbt csp = bj csp . bt csp

bj :: CSP -> Tree (State, ConflictSet) -> Tree (State, ConflictSet)
bj csp = foldTree f
  where f (s, cs) ts
          | isConflict cs  = mkTree (s, cs)  ts
          | isConflict cs' = mkTree (s, cs') []  -- plug first leak
          | otherwise      = mkTree (s, cs') ts
              where cs' = evalSet (combine (map label ts) []) -- plug second leak
```

Fig. 11. Two implementations of conflict-directed backjumping.

be a node at level l ($1 \leq l \leq m$) with a conflict set CS such that $l \notin CS \subseteq \{1, \ldots, l\}$. Then CS is also a conflict set for the parent of s, namely

$$p = (v_{l-1} := y_{l-1}, \ldots, v_1 := y_1)$$

Proof

Consider any CSP solution $(v_m := x_m, \ldots, v_1 := x_1)$. By definition of conflict set, $\exists i \in CS$ such that $x_i \neq y_i$. Since $l \notin CS$, it must be the case that $i \leq (l-1)$. Hence CS fulfills the definition of a conflict set for p as well. □

Function `bj0` in Figure 11 is a lazy bottom-up algorithm that applies the two lemmas to compute internal node conflict sets from a tree that has been (lazily) annotated with an initial labeling. At each parent node that does not already have a conflict set, `bj0` calls `combine` to build one. Function `combine` inspects the conflict sets of the children in turn. If it finds a child to which Lemma 2 applies it immediately returns this child's conflict set for use in the parent. If no such child is found, but every child has a conflict set, it applies Lemma 1. Under lazy evaluation, the subtrees corresponding to the remaining children are never explored. The combination of `bj0` with `bt` is commonly referred to as *conflict-directed backjumping* (CBJ) (or just backjumping) in the literature.

As an example, consider again the subtree of `(queens 6)` shown in Figure 6 after

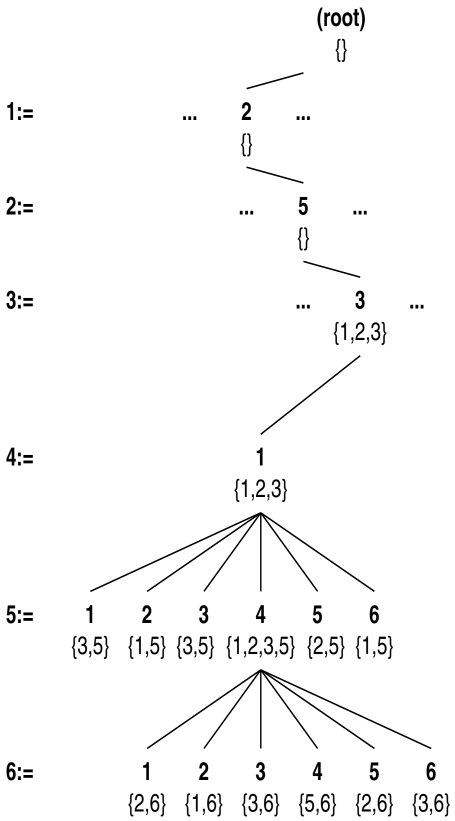

Fig. 12. Same portion of search tree for `queens 6` as in Figure 9, annotated with conflict sets as computed by `bj`. Nodes to the right of `4 := 1` have been pruned away.

queens have been placed in columns 1–3. Inspection of the corresponding board position shows that this state cannot be extended to a solution, because the queens in columns 5 and 6 are already constrained to lie in the same row, regardless of which row we choose for the queen in column 4. Figure 12 shows how backjumping takes advantage of this fact to avoid repeated placement attempts for columns 5 and 6. After all conflict sets in the left-hand subtree at level 6 have been calculated, Lemma 1 can be applied to calculate a conflict set of $\{1, 2, 3, 5\}$ for the parent node with last assignment `5 := 4`. After the other conflict sets of this node's siblings are computed, Lemma 1 can be applied again to calculate a conflict set of $\{1, 2, 3\}$ for its parent node, which has last assignment `4 := 1`. Since $4 \notin \{1, 2, 3\}$, Lemma 2 can be applied, to give the same conflict set to the parent node, which has assignment `3 := 3`. The remaining children of this last node need not be explored.

This algorithm works correctly for *any* initial conflict set labeling, but it is most effective when the conflict sets are small and contain low-numbered variables, because this increases the number of levels for which Lemma 2 can be applied. This is why we use *earliest* inconsistent pairs to represent consistency conflicts. CBJ is the cornerstone of many newly-developed algorithms (Frost, 1997). In its usual imperative formulation this algorithm is notoriously difficult to understand or prove correct. We have relied on the analysis of Caldwell *et al.* (1997) for our understanding of conflict sets, but we are unaware of any previous description of the algorithm as a form of labeling.

Although `search bj0bt` behaves just like imperative CBJ in the sense that it performs the same number of consistency checks, it has two unfortunate space leaks. The first leak occurs because the pruning phase must keep a pointer to each node's list of children until that node's conflict set has been computed, but that computation may generate a substantial part of the subtree rooted at the node. Even if most of the subtree is eventually pruned, its transient memory requirements can be exponentially large. We can plug this leak, at the cost of some loss in modularity, by adding additional pruning into the labeler itself: if a conflict set is computed for a node, we immediately remove the node's children. The second leak occurs because the thunk returned by `combine` at a given node may retain pointers to a substantial part of the subtree rooted at that node. Since the resulting conflict set will definitely be demanded further up in the tree, we can plug this leak in a straightforward way by forcing evaluation of `combine`'s result. Function `bj` shows the final code after both leaks have been plugged.

8 Backmarking

Ordinary backtracking provides one way to generate conflict sets, but it is not necessarily the best way. The `bt` labeler works by checking each assignment against all previous assignments in its state. Although this approach checks the overall consistency of each partial state only once, it can still perform many duplicate pairwise consistency checks because all the subtrees of a given node are identical except for the assignments at their roots. Consider a node s at level l, and consider any descendent s' of s other than an immediate child. In checking the consistency of s', pairwise checks will be made between its last assignment and all the assignments in s at levels less than or equal to l. These checks will be duplicated for the corresponding descendents of *every* sibling of s (unless, of course, they had an inconsistent ancestor and have been pruned away). For an example, compare the two nodes shown on level 6 of Figure 9 having last assignment `6 := 6`. To generate their conflict sets, `bt` makes the same three comparisons in each case, namely with `1 := 2`, `2 := 5`, and `3 := 3`, before encountering a conflict.

An alternative approach is to *cache* the results of such consistency checks so they can be reused for each sibling; this should reduce the total number of consistency checks at the cost of the space needed for caching. We annotate each node with a cache to store information about inconsistencies between that node's state and the assignments made in its descendents. Each cache is organized as a table having an entry for every possible assignment of the thus-far unassigned variables. If the assignment would cause a conflict with an already-assigned variable, the cache entry contains a conflict set; otherwise, it contains the empty set. Each node's cache is a

Variable	Value					
	:= 1	:= 2	:= 3	:= 4	:= 5	:= 6
4	{}	{1,4}	{2,4}	{3,4}	{1,4}	{}
5	{3,5}	{1,5}	{3,5}	{}	{2,5}	{1,5}
6	{2,6}	{1,6}	{3,6}	{}	{2,6}	{3,6}

Table 1. Fully-evaluated cache table corresponding to node at level 3 of `queens 6` search tree in Figure 9.

refinement of its parent's cache, with a cache at any given level containing complete consistency information about assignments at the next level, and partial information about assignments at lower levels. As an example, Table 1 shows the cache contents for the node at level 3 of Figures 6 and 9. The non-empty entries in the cache can be used to generate all the conflict set annotations in Figure 9 that involve variable 1, 2, or 3; that includes all the annotations at level 4, and most of those at levels 5 and 6. Note also how the rows of the cache table correspond with columns 4-6 of the inset diagram in Figure 6; in effect, caches for the n-queens problem describe threatened positions in unassigned columns.

Figure 13 shows an algorithm implementing cache-based labeling, based on an implementation of caches shown in Figure 14. Function `augmentConflicts` computes the cache contents for a node based on the node's assignment and the node's parent's cache. To do this, it maps `extendCS` over each cache entry. If the parent's cache already records a conflict set for the future assignment, that set is inherited by the current cache; otherwise a conflict check is performed and the result (an earliest conflict set or $\emptyset$) is recorded. Function `storeConflicts` applies `augmentConflicts` to each node in a tree in top-down fashion. Once the tree has been annotated with cache tables, `extractConflicts` is used to extract the conflict sets for the next unassigned variable at each node and distribute them over the node's children; the resulting tree of conflict sets is then zipped together with the state labels from the original tree. The final annotated tree is identical to that produced by `bt`.

Caches are implemented as lists of lists.[3] As usual, we rely on lazy evaluation to avoid building the tables or their contents unless they are needed. So most of the tables remain unbuilt, and the actual order in which consistency checks is performed is similar to `bt`. The important point is that, because many of a node's table entries are inherited from its parent's table, all duplicate consistency checks are avoided.

As before, we obtain a complete solver by using `bm` as the `labeler` parameter to `search`. Bacchus and Grove (1995) made the somewhat surprising discovery that this lazy caching algorithm is equivalent (in terms of consistency checks made) to a standard imperative algorithm called *backmarking*.

[3] A two-dimensional array or an array of arrays would be more obvious implementations, but they turn out not to not perform as well under the `ghc` compiler.

```
bm :: Labeler
bm csp = extractConflicts . storeConflicts csp

storeConflicts :: CSP -> Tree State -> Tree (State,Cache ConflictSet)
storeConflicts csp = inhTree f (undefined,undefined)
     where f (_,tbl) s = (s,augmentConflicts csp tbl s)

augmentConflicts :: CSP -> Cache ConflictSet -> State -> Cache ConflictSet
augmentConflicts csp@CSP{rel=rel} parentTbl s
    | isEmptyState s = initCache csp emptySet
    | otherwise = mapCache extendCS tbl
            where tbl = thinCache parentTbl (var lasta)
                  extendCS :: Assignment -> ConflictSet -> ConflictSet
                  extendCS a cs
                      | isConflict cs = cs
                      | rel lasta a   = emptySet
                      | otherwise     = listToSet [var lasta, var a]
                  lasta = lastAssignment s


extractConflicts :: Tree (State,Cache ConflictSet) -> Tree (State,ConflictSet)
extractConflicts t = zipTreesWith g t t'
  where t' = distrTree f emptySet t
        f (s,tbl) = lookupCache tbl (nextVar s)
        g (s,_) cs = (s,cs)
```

Fig. 13. Backmarking.

```
data Cache a = Cache [(Var,[a])]

initCache :: CSP -> a -> Cache a
initCache CSP{vars=vars,vals=vals} i = Cache (zip [1..vars] (repeat row))
  where row = take vals (repeat i)

thinCache :: Cache a -> Var -> Cache a
thinCache (Cache cache) var0 = Cache [(var,row) | (var,row) <- cache, var /= var0]

mapCache :: (Assignment -> a -> a) -> Cache a -> Cache a
mapCache f (Cache cache) =
    Cache [(var, newRow var row) | (var,row) <- cache]
         where newRow var row =  [ f (var := val) a | (val, a) <- zip [1..] row ]

lookupCache :: Cache a -> Var -> [a]
lookupCache (Cache cache) var = val
   where Just val = lookup var cache

getCache :: Cache a -> [(Var,[a])]
getCache (Cache cache) = cache
```

Fig. 14. Caches.

Of course, the decrease in checks comes at the cost of increased space requirements. For a problem with m variables and n values, each cache requires up to nm entries, each of which records at most 2 conflicting variables; since there can be up to m caches live at any one time, the total cost is $O(m^2n)$ cells. However, even for large problems, this is unlikely to be a significant limitation.

```
mfc :: Labeler
mfc csp = mfc' csp . storeConflicts csp

mfc' :: CSP -> Tree (State,Cache ConflictSet) -> Tree (State,ConflictSet)
mfc' csp t = zipTreesWith f (extractConflicts t) (mapTree (wipedDomain csp) t)
               where f (s,cs) cs' | isConflict cs = (s,cs)
                                  | otherwise     = (s,cs')

wipedDomain :: CSP -> (State, Cache ConflictSet) -> ConflictSet
wipedDomain CSP{vars=vars} (s,tbl)
  | null wipedDomains = emptySet
  | otherwise =  intersectSet (unionCS (head wipedDomains))
                             (listToSet (map var (assignments s)))
    where wipedDomains :: [[ConflictSet]]
          wipedDomains = [css | (v,css) <- getCache tbl, all isConflict css]
```

Fig. 15. Minimal forward checking.

9 Forward Checking

The cache tables built for backmarking can be further exploited to avoid still more consistency checks. Suppose that the table for some node s contains a row, corresponding to an as yet unassigned variable, in which every entry contains a conflict set. Then it is evident that node s can never be extended to a solution, because the assignments in s rule out all possible values for the future variable. (As an example, consider the inset diagram in Figure 6; if we add a queen at position `4 := 6`, this will rule out the sole remaining possible assignment for column 6, namely `6 := 3`, regardless of what happens in column 5.) Therefore, there must exist a non-empty conflict set for s. By labeling s with such a set, we can avoid further search in the subtree rooted at s. This technique has been called *domain wipeout* (Bacchus & Grove, 1995). The combination of domain wipeout with `bm` labeling corresponds to the well-known imperative algorithm called *forward checking*. Because our cache table construction is lazy, we have actually rediscovered ("for free") *minimal (*or *lazy) forward checking*, itself a recent discovery in the imperative literature (Dent & Mercer, 1994).

Figure 15 shows code for implementing domain wipeout in combination with back marking. The use of `storeConflicts` and `extractConflicts` is just as in backmarking, but we also keep the cache-annotated tree to use as input to `wipedDomain`. Because of laziness, the search for an empty domain is only performed if backmarking fails to find a conflict set.

To gather a list of `wipedDomains` and test whether it is non-empty is straightforward. The interesting question is what conflict set to assign to the node s if domain wipeout has occurred. Since it is always valid to throw additional variables into a non-empty conflict set, we could just use the complete set of variables assigned by s. But it is better to use the smallest available conflict sets based on the available information, because this can increase their utility for other algorithms (e.g., backjumping). In this case, the cache table row for a wiped-out domain records which existing assignment rules out each possible value for that domain. The union of the variables in these assignments (restricted to the variables assigned by s) is a valid conflict set for s, since any solution must assign a different value to at least one of them. If there is more than one wiped-out domain, we could compute a conflict set from any one of them; for simplicity and to limit computation, `domainWipeOut` just chooses the first.

10 Mixing and Matching

A major advantage of our declarative approach is that we can trivially combine algorithms using function composition, so long as they take a consistent view of conflict set annotations. For example, we can describe a labeler that combines minimal forward checking and backjumping in a single line:

```
bjmfc csp = bj csp . mfc csp
```

Imperative forward checking is traditionally described as filtering out all the conflicting values from the domains of future variables; this makes it hard to explain how it can be profitably combined with backjumping, since the latter would seem to have no information on which to base backjumping decisions. Our viewpoint is that forward checking is just a more (time-)efficient way of generating conflict sets, which makes the combination perfectly reasonable. Although backjumping has previously been combined with *strict* forward checking (Prosser, 1993b), to our knowledge it has never been combined with *minimal* forward checking.

Similarly, the combination of backmarking and backjumping

```
bjbm csp = bj csp . bm csp
```

is tricky to implement correctly in an imperative setting (Kondrak, 1994), but is simple for us, and turns out to do fewer consistency checks on `queens` than any of our other fixed-variable-order algorithms (see Table 2 in Section 12).

Once problem-specific value ordering heuristics are introduced, many more possibilities for new algorithm design open up. Since the best combination of algorithm features tends to depend on the particular problem at hand, it is important to be able to experiment with different combinations; our framework should make this extremely easy.

11 Dynamic Variable Ordering

All the algorithms described so far have used a fixed order for choosing the next variable to assign to at each tree level. We might hope to do a better job in choosing that variable by using information gathered during the search. Techniques for doing this are known as *dynamic variable ordering (DVO)* heuristics. Such heuristics can be implemented by feeding information from later stages of the search pipeline back into the generation of the search tree. A lazy framework that supports this approach is shown in Figure 16. Since the generation of the search tree is now dependent on later stages of the pipeline we have to be careful not to demand nodes in the tree before they are generated. The laziness issues involved are subtle,

```
type DVOParams a = (CSP -> Tree (State,a) -> Tree (State,ConflictSet),
                   CSP -> a -> State -> Var,
                   CSP -> a -> State -> a)

searchDVO :: DVOParams a -> CSP -> [State]
searchDVO (relabeler,selector,prelabeler) csp =
          (solutions . (relabeler csp) .
                       mkSearchTreeDVO (prelabeler csp) (selector csp)) csp

mkSearchTreeDVO :: (a -> State -> a) -> (a -> State -> Var) -> CSP -> Tree (State,a)
mkSearchTreeDVO prelabeler selector csp = initTree mk (root_s,root_a)
  where  mk (s,a) = [(newNextVar s' (selector a' s'),a') |
                               s' <- extensions csp s,
                               let a' = prelabeler a s']
         root_a = prelabeler undefined root_s
         root_s = emptyState csp
```

Fig. 16. Dynamic variable ordering.

but by encapsulating them in the implementation of the generator function, we have made it easy to add new heuristics.

The search tree generator `mkSearchTreeDVO` is parameterized by a `prelabeler` transform and a `selector` function. The prelabeler is applied to the state tree to produce an annotated tree; the selector uses the states and annotations to choose the next variable to search on from among those still unassigned. In order to avoid cyclic dependencies, the prelabeler is required to operate in top down fashion: it must compute the annotation for a node based solely on the node's state and its parent's annotation. The result of the selection step is recorded by reordering the list of unassigned variables using the `newNextVar` function from Figure 1.

The `searchDVO` function is essentially similar to ordinary `search`, except that it uses `mkSearchTreeDVO`, and it allows the search labeling function, here called a `relabeler`, to make use of the annotations built by the prelabeler. A complete solver is obtained by applying `searchDVO` to a triple `DVOParams`, consisting of a relabeler, selector, and prelabeler.

One common and practical DVO heuristic is called *fail first*; it always picks the variable with the smallest remaining domain (i.e. the smallest number of possible value assignments). In the event of a tie, the domain with lowest-numbered variable is picked. For example, in the state shown in the inset diagram in Figure 6, after the assignment `3 := 3`, the domain for column 4 contains two values, and the domains for 5 and 6 contain one value each; the fail first heuristic will therefore pick column 5 as the next variable to try. The rationale for this heuristic is that it encourages earlier identification and pruning of conflicted nodes.

Figure 17 shows a number of possible implementations of this heuristic. All of them use `augmentConflicts` (Figure 13) as the prelabeling function, which annotates each node with a cache of future conflict set information, just as in backmarking. We show three different possible selector functions for calculating the smallest remaining domain. Although all three selectors compute the same answer, and are about equally efficient in practice, they exhibit subtle differences in laziness, which

```
failFirst0 :: Cache ConflictSet -> State -> Var
failFirst0 tbl _ = var
  where (var,_) = foldr1 smallerDomain sizedDomains
        smallerDomain a@(_,asize) b@(_,bsize) = if asize <= bsize then a else b
        sizedDomains = [(var,length (filter (not . isConflict) css))
                                            | (var,css) <- getCache tbl]

ff0 :: DVOParams (Cache ConflictSet)
ff0 = (const extractConflicts,const failFirst0,augmentConflicts)

ff0solver :: CSP -> [State]
ff0solver = searchDVO ff0

failFirst :: Cache ConflictSet -> State -> Var
failFirst tbl _ = var
  where (var,_) = foldr1 smallerDomain sizedDomains
        smallerDomain a@(_,asize) b@(_,bsize) = if asize `nleq` bsize then a else b
        sizedDomains = [(var,nlength (filter (not . isConflict) css))
                                            | (var,css) <- getCache tbl]

ff :: DVOParams (Cache ConflictSet)
ff = (const extractConflicts,const failFirst,augmentConflicts)

failFirst1 :: Cache ConflictSet -> State -> Var
failFirst1 tbl _ = var
  where (var,_) = smallestDomain sizedDomains
        smallestDomain domains =
          case emptyDomains of
            d:_ -> d
            []  -> smallestDomain (map f domains)
                     where f (var,n) = (var,npred n)
          where emptyDomains = filter (isZ . snd) domains
        sizedDomains = [(var,nlength (filter (not . isConflict) css))
                                            | (var,css) <- getCache tbl]

ff1 :: DVOParams (Cache ConflictSet)
ff1 = (const extractConflicts,const failFirst1,augmentConflicts)
```

Fig. 17. Three different implementations of fail-first ordering.

are reflected in the numbers of consistency checks they perform. The first selector, `failFirst0`, scans each row of the cache table annotation, calculating the integer representing the size of the corresponding domain (i.e. the number of values for which no conflict set is recorded), and selecting the smallest domain accordingly. Although `failFirst0` calculates the correct result in a straightforward fashion, searches that use it perform more consistency checks (see Table 2 in Section 12) than imperative implementations of fail first described in the literature (Bacchus & Grove, 1995). This is because the published algorithms determine the *smallest* remaining domain without actually determining the exact size of that domain. We can achieve the same effect by calculating domain sizes using an implementation of *natural numbers* that supports lazy comparisons, as shown in Figure 18. Changing `failFirst0` to use `Nat` instead of `int` requires only that we change the functions used to compute and compare lengths; the result is shown as `failFirst`. The resulting consistency check counts match the literature.

Fail first uses the cache table in much the same way as domain wipeout does.

```
data Nat = Z | S Nat

nleq :: Nat -> Nat -> Bool
nleq  Z      _      = True
nleq  _      Z      = False
nleq (S n1) (S n2) = nleq n1 n2

isZ :: Nat -> Bool
isZ Z = True
isZ _ = False

npred :: Nat -> Nat
npred (S n) = n
npred Z     = error "npred"

nlength :: [a] -> Nat
nlength []     = Z
nlength (a:as) = S(nlength as)
```

Fig. 18. Natural numbers.

This similarity between forward checking and fail first is not coincidental; forward checking is essentially a limited form of fail first that deviates from the fixed variable ordering only when the smallest remaining domain is completely empty. We might expect, therefore, that adding forward checking to `ff`, by using the `DVOParams`

```
mfcff = (mfc',const failFirst,augmentConflicts)
```

would not lower the number of consistency checks required. However, as Table 2 shows, this is not true: the combination `mfcff` performs slightly fewer checks than plain `ff`. The reason for this is subtle: `mfc` performs only enough consistency checks to find an empty domain, whereas `failFirst` may also perform checks in order to determine which of two non-empty domains is smaller, even if there is a completely empty domain further down the list. Function `failfirst1` is a variant of `failFirst` that finds the smallest domain by first looking for an empty domain; if none is found, it decrements the size of each remaining (non-empty) domain, and tries again. The resulting `ff1` search incorporates all the behavior of forward checking, and performs exactly the same number of checks as the the combined algorithm

```
mfcff1 = (mfc',const failFirst1,augmentConflicts)
```

Moreover, `ff1` performs fewer consistency checks on `queens` than any fail-first variant we have discovered in the literature.

We can also combine fail first with backjumping:

```
bjff1 = (\csp -> bj csp . extractConflicts,
             const failFirst1,augmentConflicts)
```

This algorithm shows only tiny improvement over `ff1`, which is not surprising. We can view backjumping as a mechanism for compensating for a poor fixed variable order, in which heavily constrained variables appear late in the order; thus, it has little left to do after an effective dynamic ordering heuristic has been applied.

Queens	8	9	10	11	12	13
`bt`	46752	243009	1297558	7416541	45396914	292182579
`bjbt`	41128	214510	1099796	6129447	36890689	233851850
`bm`	12308	50866	220052	1026576	5224512	28405086
`mfc`	12276	51642	220745	1038129	5297651	28817439
`bjbm`	11928	49369	210210	975198	4938324	26709008
`bjmfc`	12229	51314	218907	1026826	5231284	28387767
`ff0`	12502	51856	214244	980640	4869822	25627720
`ff`	11934	49317	202593	924150	4590577	24183989
`mfcff`	11726	48487	197420	898096	4446851	23388513
`ff1`	11579	47385	191813	868409	4281753	22479211
`mfcff1`	11579	47385	191813	868409	4281753	22479211
`bjff1`	11579	47375	191776	868066	4280093	22468711
Solutions	92	352	724	2680	14200	73712

Table 2. Number of consistency checks performed by various algorithms on the all-solutions n-queens problem. Algorithms are identified by their labeler function or DVO parameter triple name.

12 Experimental Results

We have investigated the performance of the various algorithms on a number of simple problems. These include finding all solutions to the n-queens problem for $n \in \{8, \ldots, 13\}$; finding the *first* solution to the 16-queens problem; and finding the first solution to the graph coloring problem on each of four graphs—Anna, Miles250, Miles500, Miles1000—drawn from the Stanford Graph Base (Knuth, 1994).[4]

We tried the all-solutions queens problems using many different algorithms, measuring the number of consistency checks performed. We tried the other problems (and the all-solutions 12-queens problem) using a smaller selection of algorithms, measuring the number of consistency checks performed, elapsed execution time in user mode, and maximum heap memory use. The measurements were taken on a lightly loaded 400MHz UltraSPARC-II with 4GB of memory, using `ghc` (the Glasgow Haskell compiler) version 4.08, with optimization flags `-O2 -O2-for-C` and a 64MB target heap size, and `gcc` version 2.95.2, with optimization flag `-O2`.

As noted in Section 1, the number of consistency checks is a widely used metric for comparing algorithm efficiency in a machine- and implementation-independent fashion. Moreover, for the all-solutions n-queens problem, consistency check counts are often used to confirm that code actually implements the algorithm that it purports to. Table 2 gives precise counts for those problems. For the remaining problems, Figure 19 shows consistency check counts and normalized execution times,

[4] For each of the graph coloring problems, we set the number of available colors (i.e. the domain of allowed values) to the minimum possible number, as obtained from the literature.

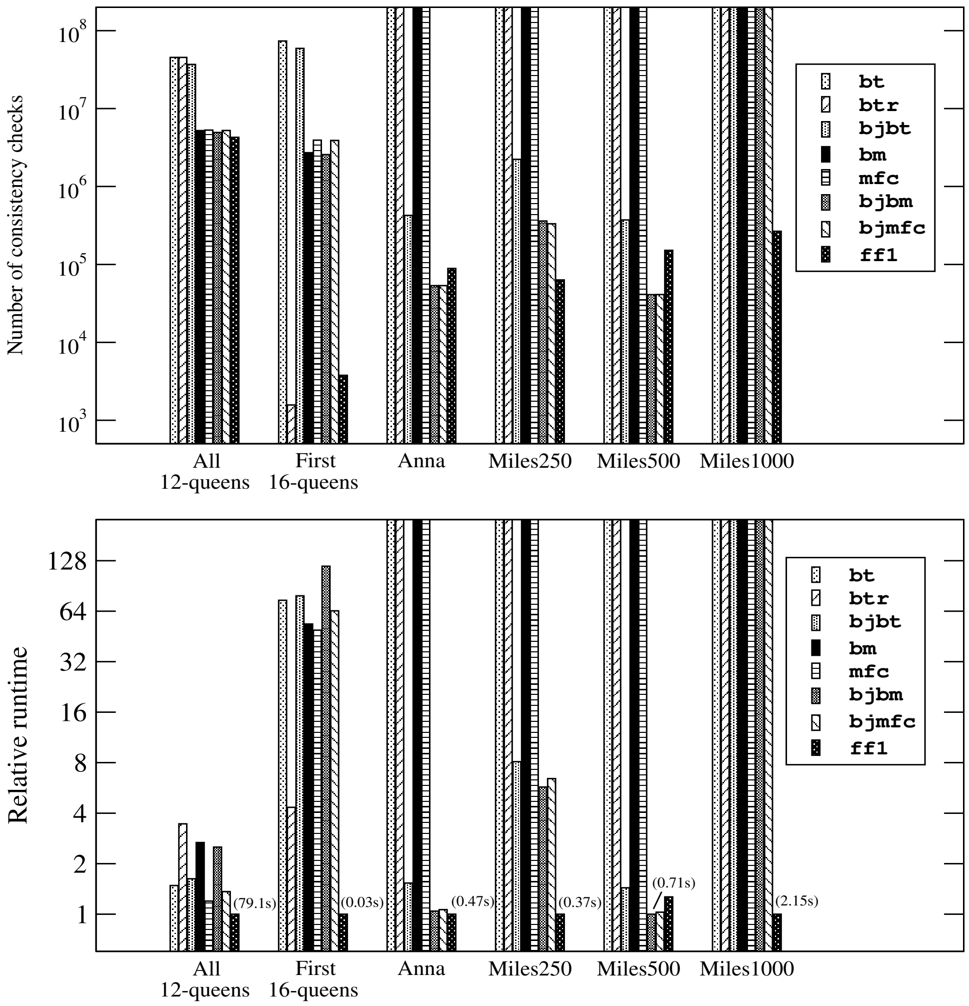

Fig. 19. Consistency checks and relative runtimes of various problem/algorithm combinations. Algorithms are identified by their labeler function or DVO parameter triple name. Runtimes for each problem are relative to the fastest algorithm for that problem, whose absolute time is indicated in parentheses. Vertical scales are logarithmic. Bars reaching to the top of the graph correspond to combinations that failed to complete within 24 hours.

both on a logarithmic scale. Each problem/algorithm combination was allowed to run for up to 24 hours; many did not terminate.

It is clear that choice of algorithm can have a significant impact on the time and number of checks required to solve these problems. Check counts correlate only moderately well with times, but the ranking of algorithms from fastest to slowest for each particular problem is roughly the same for both metrics. Perhaps the most obvious point to be made about these data is that the results vary widely

Algorithm	All 12-queens	First 16-queens	Anna	Miles250	Miles500	Miles1000
`bt`	4	7	-	-	-	-
`btr`	23	74	-	-	-	-
`bjbt`	20	80	229	180	535	-
`bm`	27	140	-	-	-	-
`mfc`	25	131	-	-	-	-
`bjbm`	44	218	1496	1390	2227	-
`bjmfc`	43	214	1501	1414	2233	-
`ff1`	24	138	2190	1497	3082	6088
variables	12	16	138	128	128	128
values	12	16	11	8	20	42

Table 3. Maximum heap memory (in kilobytes) used by problem/algorithm experiments. Algorithms are identified by their labeler function or DVO parameter triple name. Missing entries correspond to experiments that failed to complete within 24 hours.

among the different problems, confirming the need for experimentation to find the best algorithm for a particular problem, or even problem instance. However, a few general conclusions can be drawn. Fail-first dynamic variable ordering (`ff1` and its variants) is usually the best algorithm, whether measured by time or check count, sometimes by orders of magnitude. On the queens problems, the algorithms based on caching (`bm`, `mfc`, `ff1`, and their variants) make significantly fewer checks than those that are not; they do not always run faster, however, because of the overhead of maintaining the cache. On the graph problems, caching is not particularly useful, but backjumping (`bj`) makes a highly worthwhile addition to the fixed variable order algorithms; for example, `bjbt`, `bjbm`, and `bjmfc` all find solutions to Anna in under a second, whereas `bt`, `bm`, and `mfc` fail to find a solution in 24 hours! Finally, a good heuristic can be very helpful; the version of simple backtracking that randomizes its value ordering (`btr`) performs fewer checks than any other algorithm for finding the first solution to 16-queens, and five orders of magnitude fewer than unrandomized backtracking (`bt`).

Table 3 shows the maximum heap used by each problem/algorithm combination that completed within 24 hours. As expected, the only significant memory requirements are due to caching in large problems. For the graph problems, memory use is roughly proportional to the product (number of variables) $\times$ (number of values). As best we can tell, none of the algorithms leak memory.

Relative to conventional imperative implementations, our code is slow. To estimate the time cost of modularity and laziness, we wrote a conventional recursive version of backtracking search to report the number of solutions for the n-queens problem in Haskell, and compared the runtime with that of `bt`. On the 12-queens problem, the modular version runs almost ten times slower than the recursive formulation (117.3 s vs. 12.3 s). Further, informal experiments suggest that most of this slowdown is due to the need to build, read, and eventually garbage-collect tree nodes at each stage of the pipeline. To further estimate the overhead of using Haskell, we also coded the conventional recursive search algorithm in idiomatic C. The Haskell code is about three times slower than the C code (which runs in 4.4 s), probably because it displays much less locality of reference. However, even a constant factor slowdown of 30 due to implementation technology is not very significant for search problems, where a small change in algorithm can affect performance by many orders of magnitude.

13 Related Work

Hughes (1989) gives a lazy development of minimax tree search. Bird & Wadler (1988) treat the n-queens problem using generate-and-test and lazy lists. Oege de Moor (1995) describes a Gofer program that solves a certain class of optimization problems using dynamic programming; like ours, his code is structured as a lazy pipeline, but his primary aim is to demonstrate the broad applicability of a single fixed algorithm rather than to exploit easy functional composition of pipeline elements as we do.

Laziness (not in the context of lazy languages) has been used for improving the efficiency of existing CSP algorithms (Dent & Mercer, 1994; Schiex *et al.*, 1996), but as far as we know laziness has not previously been used to modularize any of the CSP algorithms presented here.

Many reformulations of standard CSP algorithms into uniform frameworks exist in the literature (Ginsberg, 1993; Tsang, 1993; Bacchus & van Run, 1995; Frost, 1997), but the frameworks typically are not modular; at best, the differences between two algorithms are highlighted by showing which lines of code have changed (Kondrak, 1994). Algorithms have been classified according to the amount of arc consistency they maintain (Kumar, 1992) or the number of nodes they visit (Kondrak, 1994). These classifications have shown that the backmarking and forward checking algorithms, which were previously thought of as being fundamentally different, actually share the same foundation (Bacchus & Grove, 1995), as we independently rediscovered (Section 9). Despite these efforts, there often remains confusion, even among experts in the field, about which algorithm a given description really implements.

Considering how long the standard algorithms have existed and how widely they are used, there have been surprisingly few published proofs of correctness. A correctness criterion for search algorithms based on soundness and completeness is presented in Kondrak (1994) and an automatic theorem prover is used to derive the algorithms in Caldwell *et al.* (1997).

As noted in Section 1, typical real-world search problems are often best handled by performing domain-specific constraint simplification before resorting to CSP-solving. A number of uniform frameworks have been developed for expressing and simplifying common kinds of constraint problems; notable examples include the family of constraint logic programming (CLP) languages (Marriott & Stuckey, 1998), the OPL language (Van Hentenryck, 1999), and Smith's algebraic theory of global search (Pepper & Smith, 1996). For problems on discrete domains, these systems ultimately rely on brute-force enumeration and testing of candidate solutions to a residual CSP, so an efficient CSP solver is an important system component. Integrating our Haskell-based solver library into one of these broader constraint-solving frameworks might therefore be quite useful, but we have not yet explored the practicality of doing so.

14 Conclusion

Expressing algorithms in a lazy functional language often clarifies what an algorithm does and what invariants it depends on. We can modularize code that traditionally has been expressed in monolithic, imperative form. Experimentation is also very easy. New combinations of algorithms, such as minimal forward checking plus conflict-directed backjumping, can be expressed in a single line of code; the equivalent algorithm in the imperative literature requires many lines of (mysterious) C or pseudocode. Despite the overheads introduced by laziness and use of Haskell, we can conduct large experiments.

The major problem of working with lazy code is difficulty in predicting run-time behavior, particularly for space. Very minor code changes can often lead to asymptotic differences in space requirements, and the profiling tools available for investigating such problems in `Hugs` and `ghc` are inadequate.

For future work, we plan to investigate further dynamic variable-reordering and value-ordering heuristics, which are at the core of current work in the AI search literature.

Acknowledgements

Sava Krstic gave very useful assistance in proving the lemmas of Section 7. Colin Runciman, Chris Okasaki, and an anonymous referee made many useful suggestions for improvements in the code and presentation. We are particularly indebted to Colin Runciman for suggesting that we view the problems of `failFirst0` as a symptom of inadequately lazy integer arithmetic.

References

Bacchus, F. and Grove, A. (1995) On the forward checking algorithm. *Proc. Principles and Practice of Constraint Programming* pp. 293–309.

Bacchus, F. and van Run, P. (1995) Dynamic variable ordering in CSPs. *Proc. Principles and Practice of Constraint Programming* pp. 258–275.

Bird, R. and Wadler, P. (1988) *Introduction to Functional Programming.* Prentice Hall.

Caldwell, J. L., Gent, I. P. and Underwood, J. (1997) Search algorithms in type theory. *Theoretical Computer Science.* (To appear in Special Issue on Proof Search in Type-theoretic Languages).

de Moor, O. (1995) A generic program for sequential decision processes. *Proc. Programming Language Implmenetation and Logic Programming* pp. 1–23.

Dent, M. J. and Mercer, R. (1994) Minimal forward checking. *Prec. of the Int'l Conference on Tools with Artificial Intelligence* pp. 432–438. IEEE Computer Society, New Orleans, Louisiana.

Frost, D. H. (1997) *Algorithms and Heuristics for Constraint Satisfaction Problems.* PhD thesis, University of California at Irvine.

Gaschnig, J. (1977) A general backtracking algorithm that eliminates most redundant tests. *Proc. International Joint Conference on Artificial Intelligence* p. 457.

Gill, A., Launchbury, J. and Jones, S. P. (1993) A short-cut to deforestation. *Proc. Functional Programming and Computer Architecture* pp. 223–232.

Ginsberg, M. L. (1993) Dynamic backtracking. *Journal of Artifical Intelligence Research* **1**:25–46.

Hughes, J. (1989) Why functional programming matters. *Computer Journal* **32**(2):98–107.

Kempe, A. B. (1879) On the geographical problem of the four colours. *American Journal of Mathematics* **2**:193–201.

King, D. and Launchbury, J. (1995) Structuring depth first search algorithms in Haskell. *Proc. ACM Principles of Programming Languages* pp. 344–354.

Knuth, D. E. (1994) *The Stanford Graph Base.* Addison Wesley. (Graph data available from `ftp://labrea.stanford.edu/pub/sgb/`.).

Kondrak, G. (1994) *A Theoretical Evaluation of Selected Backtracking Algorithms.* M.Phil. thesis, University of Alberta.

Kumar, V. (1992) Algorithms for constraint satisfaction problems: A survey. *AI Magazine* **13**(1):32–44.

Marriott, K. and Stuckey, P. (1998) *Programming with Constraints: an Introduction.* MIT Press.

Nadel, B. A. (1990) Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert* **5**(3):16–23.

Okasaki, C. (1998) *Purely Functional Data Structures.* Cambridge University Press.

Pepper, P. and Smith, D. R. (1996) A high-level derivation of global search algorithms (with constraint propagation). *Science of Computer Programming* **28**:247–271.

Peyton Jones, S. and Hughes, J. (eds). (1999) *Report on the Programming Language Haskell 98: A Non-strict, Purely Functional Language.* (Available at `http://www.haskell.org/definition/`.).

Prosser, P. (1993a) Domain filtering can degrade intelligent backtracking search. *Proc. International Joint Conference on Artificial Intellience* pp. 262–267.

Prosser, P. (1993b) Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence* **9**(3):268–299.

Schiex, T., Regin, J. C., Gaspin, C. and Verfaillie, G. (1996) Lazy arc consistency. *Proc. of AAAI* pp. 216–221.

Stergiou, K. and Walsh, T. (1999) Encodings of non-binary constraint satisfaction problems. *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence (AAAI-99)* pp. 163–168.

Tsang, E. (1993) *Foundations of Constraint Satisfaction.* Academic Press Limited.

Van Hentenryck, P. (1999) *The OPL Optimization Programming Language.* MIT Press.