MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Technical Report No. 956 August, 1988

Dependency-Directed Backtracking in Non-Deterministic Scheme

Ramin Zabih

Non-deterministic LISP can be used to describe a search problem without specifying the method used to solve the problem. We show that SCHEMER, a non-deterministic dialect of SCHEME, can support dependency-directed backtracking as well as chronological backtracking. Full code for a working SCHEMER interpreter that provides dependency-directed backtracking is included.

This is a greatly revised version of a thesis submitted to the Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science on January 2, 1987, in partial fulfillment of the requirements for the degree of Master of Science.

Draft of August 19, 1988 22:13

1. Introduction

Many problems in Artificial Intelligence involve search. SCHEMER is a non-deterministic dialect of LISP which can be used to formally define search problems. Dependency-directed backtracking is a sophisticated strategy for solving search problems. We describe how to apply dependency-directed backtracking to search problems defined in SCHEMER.

We begin by describing the SCHEMER language. We next provide an overview of dependency-directed backtracking and list its requirements. We then show how to meet these requirements in interpreting SCHEMER. This demonstrates the possibility of applying sophisticated search algorithms and search heuristics to any search problem which can be defined abstractly in non-deterministic LISP. In the final section we suggest that other search techniques which might be applied to arbitrary search problems defined in non-deterministic LISP.

2. SCHEMER

We view non-determinstic LISP as a formal language for defining search problems. More specifically, we are interested in search problems that are defined as programs in a particular dialect of non-deterministic functional LISP called SCHEMER. The language SCHEMER is based on the LISP dialect SCHEME [16]. SCHEMER consists of functional, normal-order SCHEME plus McCarthy's ambiguous operator `AMB` [13], and the special form `(FAIL)`.

The semantics of SCHEMER will be introduced informally here, mainly by example. The formal semantics of a subset of SCHEMER are given in Appendix 1 of [6]. In Clinger's terminology, SCHEMER is non-strict, singular call-by-need, and `(FAIL)` diverges.

The function `AMB` takes two arguments and non-deterministically returns the value of one of them. SCHEMER expressions involving `AMB` can thus have different possible executions, depending upon which values the `AMB`'s return. The expression `(AMB 0 (+ 1 2))` for example has two possible executions; one in which the `AMB` returns the value of its first argument, and one in which the `AMB` returns the value of its second. The possible executions give a SCHEMER expression a set of possible values. The two possible executions for the expression `(AMB 0 (+ 1 2))` produce the values 0 and 3; `(AMB 0 (+ 1 2))` therefore has the possible values $\{0, 3\}$.

Some possible executions might evaluate `(FAIL)`; such executions are said to *fail*. More precisely, they do not contribute a possible value to the set associated with the expression. For example, the expression `(AMB 0 (FAIL))` has two possible executions. However, this expression has only one possible value; the set associated with the expression is $\{0\}$. In general, the possible values of a SCHEMER expression consist of the results of all possible executions of that expression which do not fail (or diverge).

In the program below, the possible values of the expression `(ANY-NUMBER)` are the non-negative integers $\{0, 1, 2, \ldots\}$.

```
(DEFINE ANY-NUMBER
  (LAMBDA ()
    (AMB 0 (1+ (ANY-NUMBER)))))
```

The possible values of (ANY-PRIME) are the prime numbers $\{2, 3, 5, 7, 11, \ldots\}$.

```
(DEFINE ANY-PRIME
  (LAMBDA ()
    (LET ((NUMBER (ANY-NUMBER)))
      (IF (PRIME? NUMBER)
          NUMBER
          (FAIL)))))
```

There are many possible executions of the expression (ANY-PRIME); there is a possible execution for every value that (ANY-NUMBER) might return. The possible executions in which the value of (ANY-NUMBER) is not prime fail, while the possible executions in which the value of (ANY-NUMBER) is prime return that value.

The above procedure is an example of general generate and test. Many search problems can be viewed as the problem of finding possible values for an expression of the form:

```
(LET ((CANDIDATE (GENERATE)))
  (IF (TEST? CANDIDATE)
      CANDIDATE
      (FAIL)))
```

Constraint satisfaction problems [14], for example, can be written as generate and test procedures in this manner.

A SCHEMER expression can be viewed as defining a search problem. The search problem is to find the possible values of the expression, i.e., the results of all non-failing possible executions. The idea of expressing search problems in a non-deterministic language is not new; see [7] for a review of previous work along these lines. Our emphasis, however, is on applying sophisticated search algorithms to problems defined in SCHEMER. In particular, we are interested in applying dependency-directed backtracking.

3. Dependency-Directed Backtracking

Dependency-directed backtracking is a general search strategy invented by Stallman and Sussman [18]. We will define it as a technique for pruning search trees. Consider an arbitrary search tree generated by some particular search. The

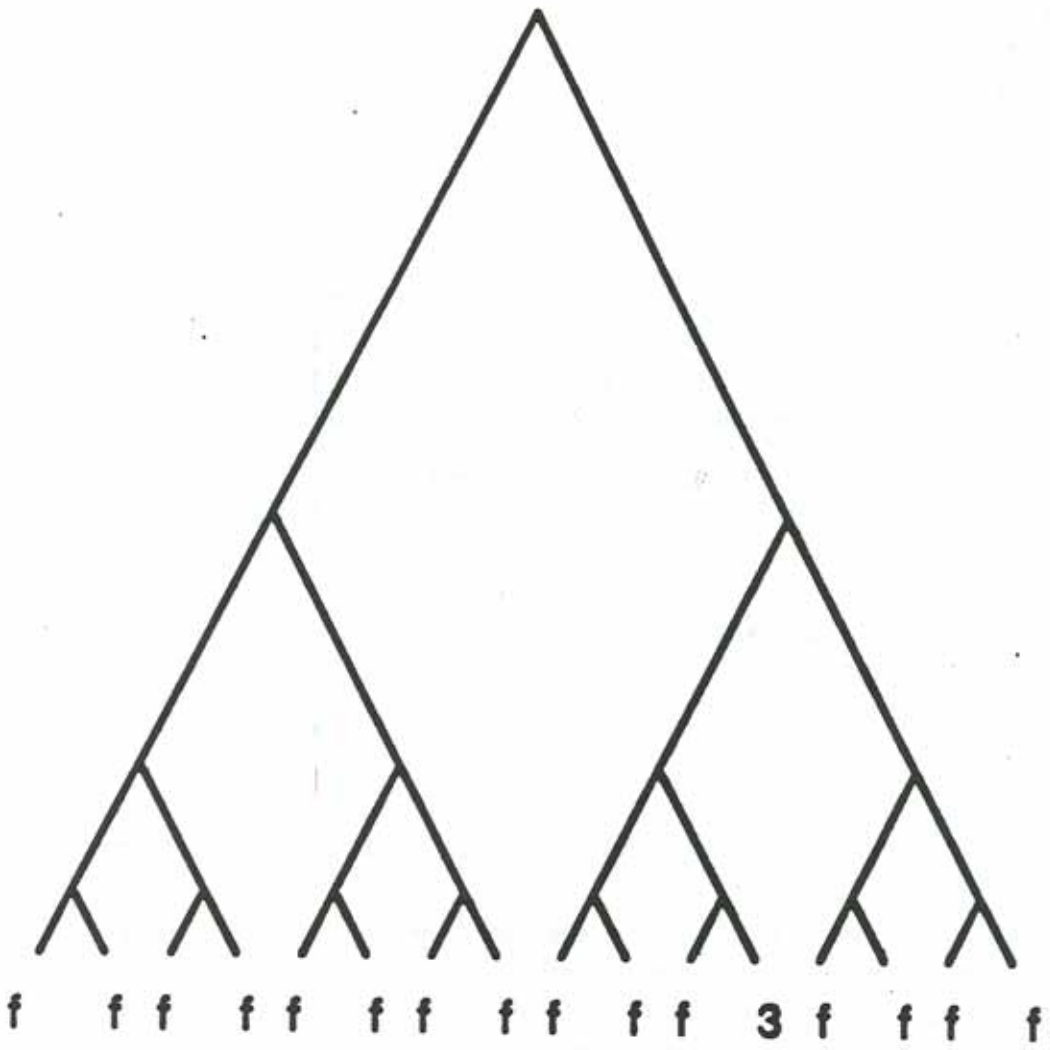

Figure 1. A search tree with one solution. Failures are labeled "f".

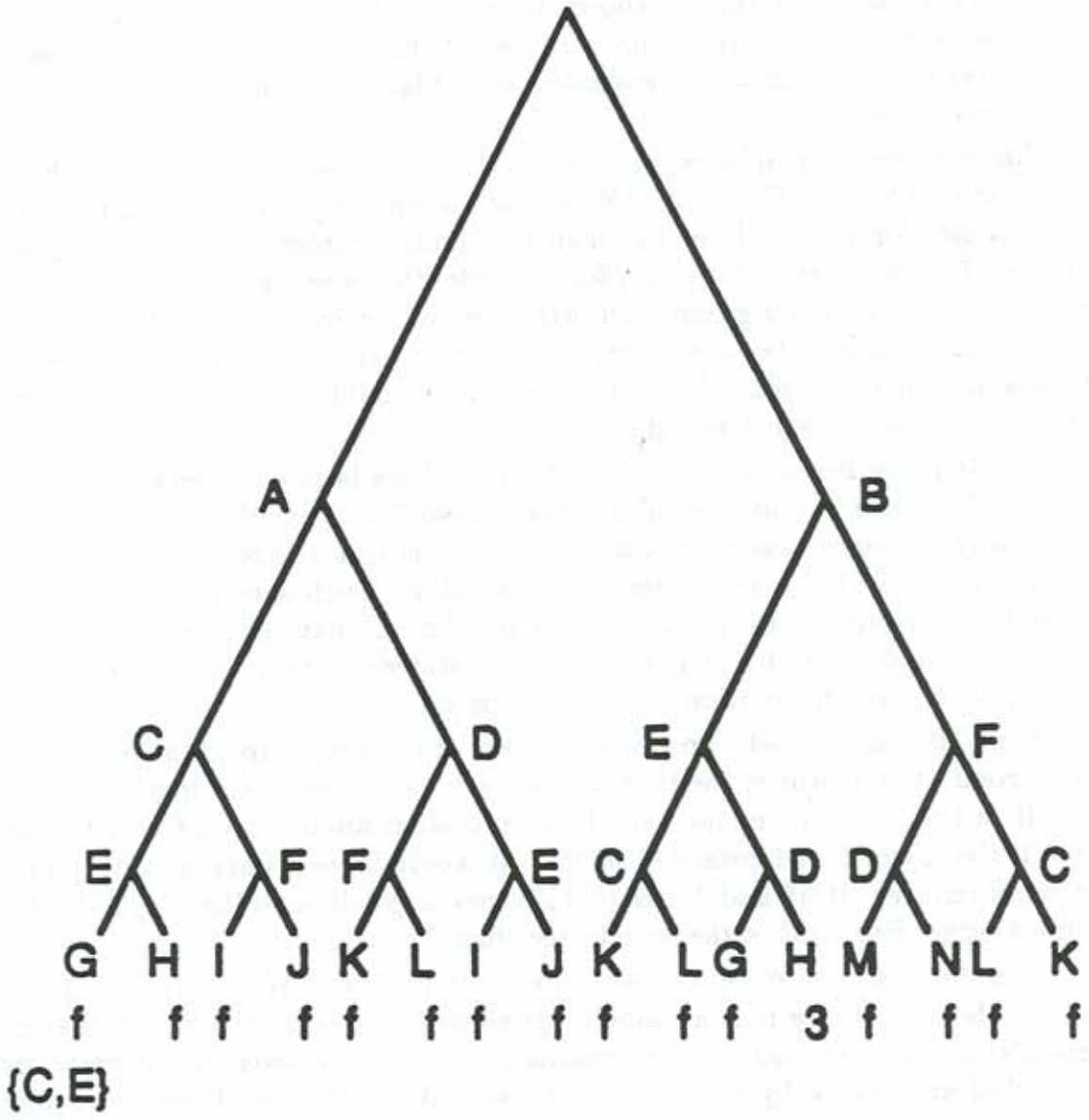

Figure 2. The same search tree after labeling and dependency analysis. Capital letters are labels. Dependency analysis is shown only for the leftmost failure.

leaf nodes of the search tree will be candidates. These candidates are either solutions to the search problem, or they are failures. Such a tree is shown in Figure 1; failures have been labeled with the letter "f".

Dependency-directed backtracking is a technique for identifying unsearched fragments of the tree which cannot contain solutions. This is done by augmenting the search tree with two pieces of information from the search problem. First, the non-root nodes must be assigned labels. Each label represents a statement that is known to be true of all leaf nodes beneath the labeled node (a label on a leaf node represents a statement that is known to true of that leaf). Second, each failing leaf node must be associated with a subset of the set of labels that appear at or above that leaf. The process of assigning sets of labels to failing leaf nodes is called *dependency analysis*.

Labeling and dependency analysis on the tree shown in Figure 1 might result in the tree shown in Figure 2. (Note that dependency analysis must produce a set of labels for each failure; to make the figure simpler, the set is only shown for the leftmost failure.) Suppose, for example, that the above tree represents the search for a coloring of a graph such that adjacent vertices have distinct colors, and suppose that n is a vertex in the graph. In this case the label A might represent the statement that n is assigned the color red. All candidate colorings at or beneath the search node labeled A would color n red.

The leftmost leaf node in the tree of Figure 2 has been assigned the set of labels $\{C,E\}$. This means that the failure was "caused" by the labels C and E. More specifically, it means that every leaf node which is beneath both a node labeled C and a node labeled E is guaranteed to be a failure. Such a set of labels is called a *nogood*. For example in a graph coloring problem C may represent the statement that p is colored red and E may represent the statement that m is colored red, and we may know that no solution can color both p and m red.[1]

Nogoods can be used to prune fragments of the search tree. In the above tree the nogood $\{C,E\}$ prunes the first and second leaf nodes (counting from the left) as well as leaf nodes nine and ten. These represent about a quarter of the search tree. If the nogood had consisted of $\{C\}$, it would have pruned about half of the entire search tree. If M and N are two distinct nogoods such that $M \subset N$, M will prune a larger fragment of the search tree than N.

More formally, let N be a nogood, i.e. a set of labels. We say that N *prunes* a node in the search tree if every label in the set N appears at or above that node. Dependency-directed backtracking maintains a set of nogoods, and never looks at nodes that are pruned by a nogood in this set. When the search process examines a leaf node that turns out to be a failure, dependency analysis is used to generate a new nogood; this is added to the set of nogoods and the process continues.

A particular method of node labeling and dependency analysis is called *sound*

[1] One need not assume that a single nogood is generated at a failure; multiple nogoods are also possible. However, they can be treated in much the same way, and so we will focus on single nogoods for simplicity.

if the nogoods associated with failures only prune failures; solutions should never be pruned. The labels above a candidate provide a set of statements about that candidate. If the candidate is a failure, dependency analysis is required to determine the labels (statements) which were responsible for the failure. This process can be viewed as localizing the failure; however, it must be done with care to avoid creating unsound nogoods. For example, if dependency analysis on the leftmost failure in Figure 2 overlooked the contribution of C, a nogood consisting of $\{E\}$ would be created, which would discard the only solution.

3.1 Selective backtracking and lateral pruning

Because of its tree-pruning, dependency-directed backtracking need not search the entire tree to find the solutions. When the search process comes to a failure, it might not backtrack to the most recent choice; dependency-directed backtracking sometimes skips over irrelevant choices. Consider the leftmost failure in Figure 2. Exhaustive left to right depth first search would backtrack to the most recent choice (which happens to be labeled E), and would next visit the second failure from the left. Dependency-directed backtracking, however, will use the nogood $\{C,E\}$ to skip back over the most recent choice. Instead, it will backtrack to the choice before that (the second most recent, which happens to be labeled C). It will thus avoid the second failure from the left entirely.

When a failure node is found in the search process one is always justified in backtracking to the first node which is above some member of the generated nogood set. The process of backtracking to some choice other than the most recent one will be called *selective backtracking*. It is easy to see that selective backtracking is a kind of pruning.

Selective backtracking accounts for some of the pruning done by dependency-directed backtracking, but not all. The nogood created at a failure is also recorded for later use and can prune parts of the search tree that are laterally distant from the original failure. (We will say that two nodes in the search tree are laterally distant if neither is a descendant of the other.) This second kind of pruning can occur when the same label appears down different branches of the search tree. In Figure 2, the labels C and E appear above the ninth failure from the left (among others); the originating failure is the leftmost one, which is laterally distant. We will call this behavior *lateral pruning*.

Some researchers have constructed systems which perform selective backtracking but not lateral pruning. In some cases lateral pruning is ineffective because a given label never appears down more than one branch of the search tree. Even if a given label can appear on more than one branch, lateral pruning may still lead to a less efficient search procedure; lateral pruning, unlike selective backtracking, requires storing and accessing a potentially large set of nogoods. On the other hand, there are cases where lateral pruning eliminates very large fragments of the search space. The next section shows that full dependency-directed backtracking,

with both selective backtracking and lateral pruning, can be applied to any search problem defined in SCHEMER.

4. Dependency-Directed Backtracking for SCHEMER

Finding the possible values for a given SCHEMER expression involves searching the possible executions for ones which do not evaluate (FAIL). The search has an associated binary search tree; each branch in the search tree corresponds to selecting either the first or second argument as the value of a particular AMB expression. The leaf nodes of the search tree correspond to possible executions; each possible execution either produces a value or fails.

Once can solve an arbitrary problem defined in SCHEMER by searching all possible executions of the given SCHEMER expression in a brute force manner. It is more difficult, however, to apply dependency-directed backtracking to such a search problem, as this requires a way of labeling the search tree and performing dependency analysis on the failures. This must be done in such a way that the induced pruning is sound; no possible values should be eliminated via pruning. Furthermore, to allow lateral pruning, it must be possible for the same label to appear down different branches of the search tree.

Recall that a label on a node in a search tree represents a statement that is true of all candidates under that node. We will label SCHEMER search trees with statements of the form "AMB-37 chooses its first argument" where AMB-37 refers to a particular occurrence of AMB in the expression. Such a statement will hold of the possible executions underneath this label. In general, to label the search tree we need to name particular occurrences of AMB in the expression. We will call an AMB expression that has been given a name, like AMB-37, a named AMB (in this case, named AMB number 37).

Figure 3 shows how dependency-directed backtracking can be used to find the possible values of the following expression.

```
(LET ((X (AMB 2 (AMB 3 4)))
      (Y (AMB 5 6)))
  (+ X Y (IF (= Y 5) (FAIL) 2)))
```

This expression has the possible values $\{10, 11, 12\}$. At the top of the figure is a new version of this expression, where the AMB's have been given names. The corresponding search tree is shown beneath.

The non-root nodes of the search tree have been labeled with statements about particular AMB's choosing their left or right arguments, and dependency analysis has been performed on the leftmost failure. The label AMB-37-L, for example, represents the statement that the AMB expression AMB-37 chooses its first (left) argument, while the label AMB-37-R represents the statement that AMB-37 chooses its second (right) argument. In this tree the failure of the leftmost node is caused by the fact that

```
(LET ((X (AMB-37 2 (AMB-38 3 4)))
      (Y (AMB-39 5 6)))
  (+ X Y (IF (= Y 5) (FAIL) 2)))
```

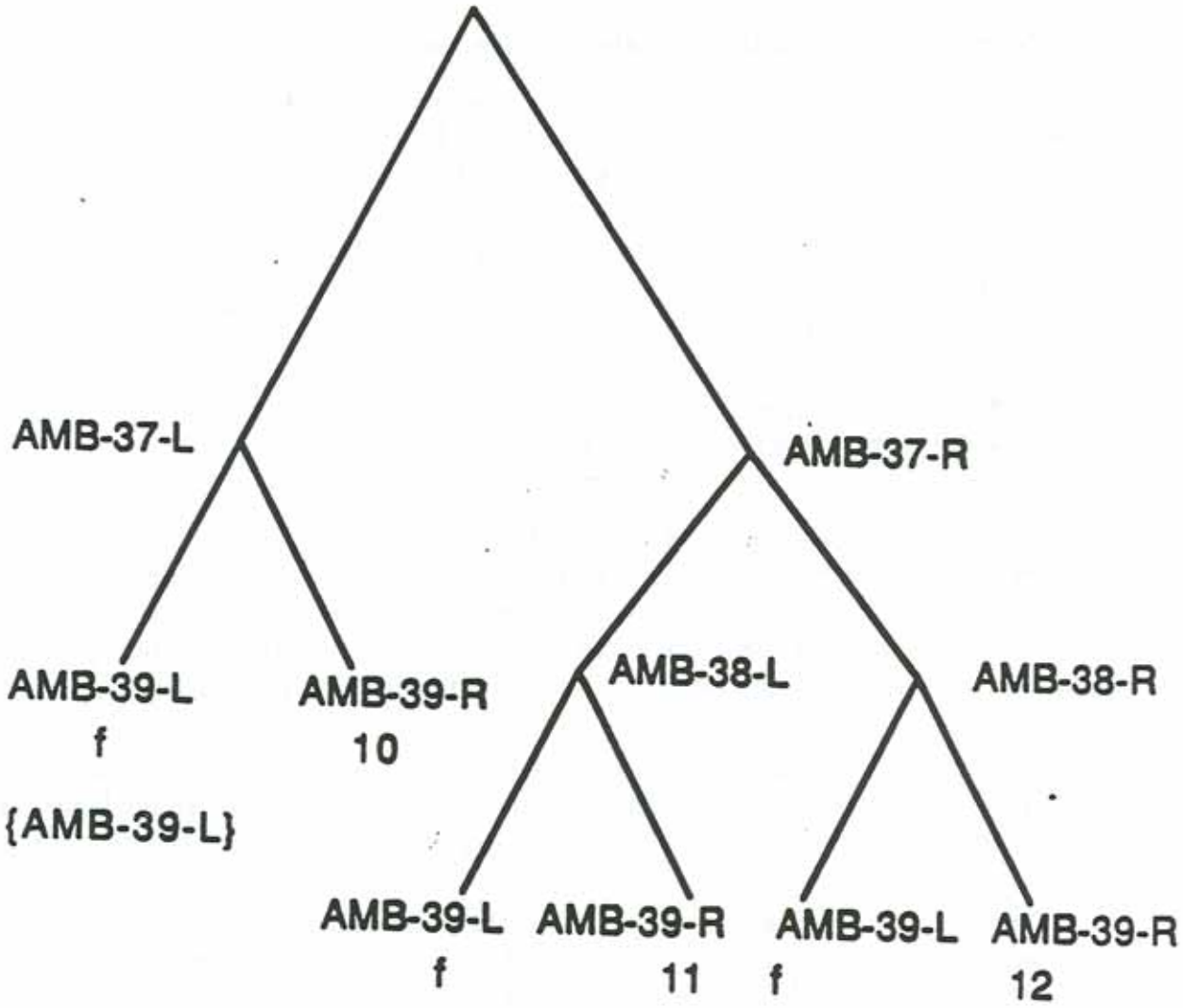

Figure 3. A SCHEMER expression with named AMB's, and its labeled search tree. Dependency analysis is shown for the leftmost failure.

AMB-39 chose its first argument. The nogood consisting of the single label AMB-39-L prunes the first, third and fifth leaf nodes. It should be clear from the SCHEMER expression that any possible execution in which AMB-39 chooses its first argument will fail.

In order to label the search tree for an arbitrary SCHEMER expression, we must first name the AMB's. This turns out to be surprisingly difficult. We will devote most of this section to solving this problem, returning to dependency analysis at the end.

4.1 Named AMB's and Equivalent Expressions

The obvious technique of naming each occurrence of AMB and then deriving labels in the search tree from these names works on the example we showed in Figure 3. However, it is unsound in general. This is because an occurrence of AMB inside a lambda expression may lead to several semantically distinct choice points. A single occurrence of AMB inside a lambda expression therefore cannot be given a single name.

For example, consider the expression

```
(LET ((BIT-GENERATOR (LAMBDA () (AMB 0 1))))
  (+ (BIT-GENERATOR) (BIT-GENERATOR)))
```

There is only one occurrence of AMB in the above expression but there are two independent choices to be made in determining a possible execution, and thus four possible executions. The above expression has the possible values $\{0, 1, 2\}$ (two of the possible executions yield the value 1). Because the single occurrence of AMB generates two independent choices, it would be unsound to give this occurrence a single name.

Our approach to this problem will be to convert a SCHEMER expression into an "equivalent" expression which is "fully named". The notion of equivalence for expressions and the notion of a fully named expression are defined more precisely below.

Formally, a named AMB is a symbol of the form AMB-n where n is a natural number. Let E be a SCHEMER expression which contains named AMB's. The set of possible values of E is defined in much the same way that the set of possible values for a SCHEMER expression are defined. However, we require that in any possible execution all occurrences of the same named AMB are consistent, i.e. they either all choose their left argument or they all choose their right argument.[2] We say that two expressions E_1 and E_2 are equivalent if they have the same set of possible values.

We can now give some examples of converting a SCHEMER expression into an equivalent expression where every AMB has been given a name. The expression

[2] This will be made more precise in Section 4.2.

```
((LAMBDA (G) (+ (G) (G))
   (LAMBDA () (AMB 0 1))))
```

is equivalent to the expression

```
(+ (AMB-40 0 1) (AMB-41 0 1))
```

Both of these expression have the possible values $\{0, 1, 2\}$. The two arguments to + represent different, independent choices, and may have different values. This example suggests that if we eliminate applications of lambda expressions by β-substitution, it is then possible to name the AMB's correctly.

However, consider the expression

```
((LAMBDA (X) (+ X X))
   (AMB 0 1))
```

which is equivalent to

```
(+ (AMB-42 0 1) (AMB-42 0 1))
```

Both of these expressions have the possible values $\{0, 2\}$. The number 1 is not a possible value because the two arguments to the primitive + must always be the same. This example suggests that one should β-substitute, but only *after* giving the AMB's names.

However, neither strategy is valid for the following expression. The 'naming before β-substitution' strategy and the 'β-substitution before naming' strategy both lead to non-equivalent expressions for

```
((LAMBDA (G N) (+ (G) (G) N N))
   (LAMBDA () (AMB 0 1))
   (AMB 2 3))
```

This expression has three independent choices and is equivalent to

```
(+ (AMB-43 0 1) (AMB-43 0 1) (AMB-44 2 3) (AMB-45 2 3))
```

As these example suggest, converting an expression into an equivalent expression where every AMB has been named is non-trivial.

Our algorithm for converting an expression into an equivalent fully named expression will be given in Section 4.4. The key idea is to interleave naming and β-substitution. More specifically, the algorithm recursively names the arguments to a function before performing β-substitution.

4.2 Contexts and Fully Named Expressions

An algorithm to name the AMB's in a SCHEMER expression must meet two soundness conditions.

Equivalence: The resulting expression must have the same possible values as the original expression.

Full naming: All occurrences of AMB in the resulting expression which correspond to a split in the search tree must be given a name.

If a SCHEMER expression satisfies the second condition, we say that it is *fully named*. We will discuss full naming first, returning to equivalence at the end of Section 4.4.

To define the notion of full naming more precisely, we need some more terminology.

Definition: A SCHEMER expression E is *deterministic* if it has exactly one possible execution.

The search tree corresponding to a deterministic expression is, obviously, trivial (as it contains only the root node). Not all deterministic expressions have a possible value; the expression (FAIL), for example, has no possible values. Any expression which does not involve AMB is deterministic. However there are expressions which contain occurrences of AMB but are nonetheless deterministic. For example the expression

```
(LAMBDA () (AMB 0 1))
```

has only one possible value: the function of no arguments which, when called, nondeterministically chooses to return either 0 or 1. The AMB in this expression does not correspond to a split in the search tree. All lambda expressions in SCHEMER are deterministic.

The application of a lambda expression, however, may be non-deterministic. For example consider the expression

```
((LAMBDA (X) (+ X (AMB 0 1))) 5)
```

This has two possible executions, depending upon which value the AMB returns, and therefore has the possible values $\{5, 6\}$.

When computing a possible value for an expression which contains named AMB's one must ensure that a named AMB, say AMB-52, behaves the same way at every occurrence: the function AMB-52 either chooses its first argument at every occurrence or chooses its second argument at every occurrence. This can be modeled by introducing the functions

```
(LAMBDA (X Y) X)
```

which we will write $\mathcal{L}$, and

```
(LAMBDA (X Y) Y)
```

written $\mathcal{R}$. We can then ensure consistency by requiring that AMB-52 either denotes $\mathcal{L}$ or denotes $\mathcal{R}$.

Definition: An *assumption* is a binding of the form AMB-n←f where f is either the function $\mathcal{L}$ or the function $\mathcal{R}$. A *context* ρ is a set of assumptions about named AMB's that does not contain two assumptions about the same named AMB, i.e. a partial map from the set of named AMB's to the set $\{\mathcal{L},\mathcal{R}\}$. If a context ρ contains an assumption AMB-n←f we will define $\rho(\text{AMB-}n) = f$. For any context ρ and expression E we define $\rho(E)$ to be the result of replacing each named AMB in E that also appears in ρ with either the function $\mathcal{L}$ or $\mathcal{R}$ as given by ρ. If a context ρ contains an assumption about every named AMB that appears in E, we will say that ρ is *complete* for E.

Definition: A SCHEMER expression E is *fully named* if for every context ρ which is complete for E, the expression $\rho(E)$ is deterministic.

To find the possible values of a SCHEMER expression, the expression is first converted to a fully named expression via the algorithm described in Section 4.4. If the expression is fully named then every AMB that corresponds to a split in the search tree will be named, and every node in the search tree can be labeled with an assumption of the form AMB-n←$\mathcal{L}$ or AMB-n←$\mathcal{R}$.

In order to ensure that every AMB occurrence that corresponds to a split in the search tree is in fact named, the naming algorithm is sometimes forced to perform infinite β-substitution. To represent such infinite β-substitution the algorithm is capable of returning an infinite expression. We handle infinite expressions with lazy S-expressions. Lazy S-expressions, which are analogous to streams [1], delay the computation of their parts until those parts must be computed. When a portion of a lazy S-expression is computed, the result is saved. Conceptually, however, the infinite expression is created in full and returned by the naming algorithm.

4.3 Syntactically Named Expressions

In general it is not possible to determine if a given expression is fully named; even if the expression is finite, there may be a fragment of the expression which contains an unnamed occurrence of AMB such that one cannot determine if that fragment will ever be executed. Fortunately, it turns out that there is a simple syntactic criterion which guarantees that an expression is fully named. This criterion is expressed in the following definition. It is important to remember that syntactically named expressions may be infinite.

Definition: A *syntactically named expression* is any one of the following:

A constant (SCHEME atom),

(FAIL) (i.e. failure),

A lambda expression of the form (LAMBDA ($x_1 \ldots x_n$) E), where the x_i's are symbols, E is an arbitrary SCHEMER expression which may contain both named and anonymous AMB's,

A primitive application (P E_1 E_2), where P is a SCHEME primitive such as + and E_1 and E_2 are syntactically named expressions,

An expression of the form (AMB-n E_l E_r) where E_1 and E_2 are syntactically named expressions,

A conditional (IF E_{pred} E_{conseq} E_{alter}), where E_{pred}, E_{conseq} and E_{alter} are syntactically named expressions.

In a primitive application the function P cannot be a lambda expression. Fully named expressions are similar to Böhm trees in the lambda calculus [2].

The reason we are interested in syntactically named expressions is that they are fully named. So if we turn a SCHEMER expression into an equivalent expression which is syntactically named, we will have succeeded in labeling the search tree while preserving the possible values of the expression. Formally, we have the following statement.

Determinism Theorem: If E is syntactically named then E is fully named. More precisely, if E is syntactically named, then for an complete context ρ the expression $\rho(E)$ is deterministic.

Proof: This theorem can be proved for finite expressions via structural induction. The theorem is true for constants and failure. As mentioned, lambda expressions are deterministic in SCHEMER, and so the theorem holds for them. The remaining cases in the definition of a syntactically named expression can be proven by assuming that the theorem holds for fully named subexpressions. To see that the theorem holds for infinite expressions consider a fully named infinite expression E with more than one possible execution. In this case there must be some finite subset of E which also has more than one possible execution. But the above induction shows that finite fully named expressions have only one possible execution. □

The naming algorithm given below will convert a SCHEMER expression into an equivalent syntactically named expression. The above theorem states that this process gives names to enough AMB's so that we can completely label the search tree for dependency-directed backtracking.

4.4 The Naming Algorithm

We are finally ready to present our naming algorithm. This algorithm converts SCHEMER expressions into equivalent syntactically named expressions. The algorithm takes as input an expression E, which may already contain some named AMB's. Selecting numbers for use in naming AMB's is done via a global name counter which is incremented every time a new name is selected. At any point in any computation the name counter will have a finite value. The potentially infinite expressions generated by this algorithm are implemented as lazy S-expressions which are expanded on demand.

ALGORITHM name(E):

1. [Name AMB's] If E is the symbol AMB, let n be the current value of the global name counter. Increment the global name counter, and return AMB-n.

2. [Constants] If E is any other atom, or `(FAIL)`, or a lambda expression, then return E.

3. [Combinations] Otherwise E is an expression of the form `(`F E_1 ... E_n`)`. Let $G =$ `name(`F`)`.

   3.1. [Simple applications] If G is either a primitive function, a named `AMB`, or `IF`, then return the expression

   `(`G `name(`E_1`)` ... `name(`E_n`))`.

   3.2. [Applying conditionals] If G is a conditional `(IF` E_{pred} E_{conseq} E_{alter}`)` then return

   `(IF` E_{pred} `name((`E_{conseq} E_1 ... E_n`)) name((`E_{alter} E_1 ... E_n`)) )`.

   3.3. [Applying `AMB` applications] If G is `(AMB-n` E_l E_r`)` then return

   `(AMB-n name((`E_l E_1 ... E_n`)) name((`E_r E_1 ... E_n`)) )`

   3.4. [Applying lambdas] If G is a lambda expression

   `(LAMBDA (`$x_1 \ldots x_n$`)` E`)`

   let E' be the result of simultaneously replacing each x_i in E with `name(`E_i`)` under the rules of β-substitution. Return `name(`E'`)`.

To see that this algorithm works, we will apply it to some examples from Section 4.1. Consider applying the above algorithm to the expression

```
((LAMBDA (G) (+ (G) (G))) (LAMBDA () (AMB 0 1)))
```

Step 3.4 will replace `G` in the body of the first `LAMBDA` with

`name(`**`(LAMBDA () (AMB 0 1))`**`)`.

By step 2 we have

`name(`**`(LAMBDA () (AMB 0 1))`**`) =` **`(LAMBDA () (AMB 0 1))`**

Replacing `G` with this produces

```
(+ ((LAMBDA () (AMB 0 1))) ((LAMBDA () (AMB 0 1))))
```

Step 3.4 then recursively names this expression, ultimately yielding

```
(+ (AMB-40 0 1) (AMB-41 0 1))
```

which is an equivalent fully named expression.

Another example was

```
((LAMBDA (X) (+ X X)) (AMB 0 1))
```

Step 3.4 will replace `X` in the body of this `LAMBDA` with `name(`**`(AMB 0 1)`**`)`. From the definition of the algorithm one can see that `name(`**`(AMB 0 1)`**`)` is an application of a named `AMB` such as `(AMB-42 0 1)`. Step 3.4 thus generates

```
(+ (AMB-42 0 1) (AMB-42 0 1))
```

which is an equivalent fully named expression.

The reader may wish to verify that our naming algorithm also produces the correct result for the other sample SCHEMER expressions given in this paper.

This naming algorithm turns a SCHEMER expression E into an equivalent fully named expression. name(E) is fully named because it is a syntactically named expression. name(E) is syntactically named because the algorithm returns either (FAIL), a lambda expression, a named AMB, an application of a named AMB, a conditional expression or an application of a primitive. In particular, the only applications in name(E) are applications of primitives.

The proof that name(E) is equivalent to E works by induction on the computation: if we assume that this correctness property holds for each recursive call then it holds for the overall algorithm. The most interesting case is step 3.4, the application of a lambda expression. β-substitution need not be sound when some argument has more than one possible execution. The following theorem, however, shows that β-substitution is sound as long as the arguments to the lambda application are fully named.

Equivalence Theorem: If E is fully named, then performing β-substitution on the expression

((LAMBDA (X) B) E)

produces an equivalent expression. (Here, B is an arbitrary SCHEMER expression, possibly containing some named choices.)

Proof: Recall that two expressions are equivalent if they have the same set of possible values. The set of possible values of any expression G is equal to the union over all complete contexts ρ for G of the set of possible values of the expression $\rho(G)$. Let C be the application shown above, and let C' be the result of the substitution (i.e., replacing occurrences of X within B by E under the rules of β-substitution). To prove the above theorem, therefore, it suffices to show that for each complete context ρ for C the expression $\rho(C)$ is equivalent to $\rho(C')$. Since E is fully named, $\rho(E)$ must be deterministic. The theorem now follows from the fact that in SCHEMER β-substitution is sound if the argument to the lambda expression is deterministic. □

The above theorem depends on the fact that the semantics of SCHEMER are based on normal order interpretation. If applicative order semantics are used then β-substitution is not sound even in the case where the arguments are deterministic. The expression

```
((LAMBDA (F) NIL) ((LAMBDA (X) (X X)) (LAMBDA (X) (X X))))
```

will diverge in an applicative order interpreter, but has the value NIL after substitution. This demonstrates that β-substitution is unsound for deterministic applicative order LISP.

4.5 Dependency Analysis

The purpose of dependency analysis is to generate a sound nogood from a failure. In SCHEMER, a failure is a possible execution of an expression E which evaluates (FAIL). The labeling process we have described names the AMB's in E. The labels that appear above a failure will be assumptions; the assumptions above

a failure form a context ρ. This context determines a possible execution for E which fails.

Recall from Section 3 that dependency analysis must produce a subset of the labels that appear above the failure which forms a nogood. If we can find a subset of the assumptions in ρ whose presence in any context guarantees that E fails, we will have a nogood. To be more precise about this, we again need some terminology.

Definition: A partial context ρ is said to *determine* an expression E if for all contexts ρ' containing ρ the expression $\rho'(E)$ is equivalent (has the same possible values) as the expression $\rho(E)$.

Consider a particular failing leaf node in the search tree for a search problem defined by a fully named SCHEMER expression E. Such a failing leaf node is associated with a set of assumptions, i.e. a context, ρ such that expression $\rho(E)$ has no possible values. (We will say that $\rho(E)$ has no possible values if every possible execution of $\rho(E)$ fails.)

Dependency analysis must find a subset of ρ which "caused" the failure. It suffices to find a subset ρ' of ρ such that ρ' determines E; if ρ' determines E then, by the definition of determination, $\rho(E)$ must be equivalent to $\rho'(E)$ and any context which contains ρ' will lead to failure. So any subset ρ' of ρ such that ρ' determines E will be a valid nogood. Of course the subset ρ' should be made as small as possible; small nogoods prune large fractions of the search space.

To make this more concrete, let E be the SCHEMER expression shown in Figure 3. The leftmost failure is the result of the context $\rho = \{\texttt{AMB-37-L}, \texttt{AMB-39-L}\}$. The subset $\{\texttt{AMB-39-L}\}$ actually determines the expression; if a context ρ' contains $\{\texttt{AMB-39-L}\}$ then the expression $\rho'(E)$ has no values.

There is a simple recursive algorithm for dependency analysis in SCHEMER. For any fully named expression E and complete context ρ we compute a subset $\texttt{just}(E, \rho)$ of the context ρ such that $\texttt{just}(E, \rho)$ determines ρ. If $\rho(E)$ has no values then $\texttt{just}(E, \rho)$ is a nogood which can be used to prune the search.

ALGORITHM $\texttt{just}(E, \rho)$:

1. [Terminals] If E is a constant, `(FAIL)`, or a lambda expression then $\texttt{just}(E, \rho)$ is the empty set $\emptyset$.

2. [Choices] If E is an application of a named `AMB`, `(AMB-`n E_l E_r`)`, then
$$\texttt{just}(E, \rho) = \begin{cases} \{\texttt{AMB-}n\leftarrow\mathcal{L}\} \cup \texttt{just}(E_l, \rho) & \text{if } \rho(\texttt{AMB-}n) = \mathcal{L}; \\ \{\texttt{AMB-}n\leftarrow\mathcal{R}\} \cup \texttt{just}(E_r, \rho) & \text{if } \rho(\texttt{AMB-}n) = \mathcal{R}. \end{cases}$$

3. [Conditionals] If E is a conditional `(IF` E_{pred} E_{conseq} E_{alter}`)` then

   3.1. [Failure] If $\rho(E_{pred})$ fails, then $\texttt{just}(E, \rho) = \texttt{just}(E_{pred}, \rho)$.

   3.2. [False] If $\rho(E_{pred})$ evaluates to[3] `NIL`, then
   $$\texttt{just}(E, \rho) = \texttt{just}(E_{pred}, \rho) \cup \texttt{just}(E_{alter}, \rho).$$

[3] formally, we should say "has the value, under the standard normal-order LISP evaluation rules". $\rho(E_{pred})$ might be something like `((LAMBDA (X Y) X) NIL)`.

3.3. [True] Otherwise,

$$\texttt{just}(E, \rho) = \texttt{just}(E_{pred}, \rho) \cup \texttt{just}(E_{conseq}, \rho).$$

4. [Primitive application] Otherwise, E is a primitive application $(P\ E_1\ E_2)$. If $\rho(E_1)$ has the value `(FAIL)`, then $\rho(E)$ also fails and $\texttt{just}(E, \rho) = \texttt{just}(E_1, \rho)$. Similarly, if $\rho(E_2)$ has the value `(FAIL)` then $\rho(E)$ fails and $\texttt{just}(E, \rho) = \texttt{just}(E_2, \rho)$. Otherwise

$$\texttt{just}(E, \rho) = \texttt{just}(E_1, \rho) \cup \texttt{just}(E_2, \rho).$$

The reader may want to verify that this algorithm produces the nogood {`AMB-39-L`} for the leftmost failure in Figure 3.

The last clause in the algorithm makes use of the fact that primitives in SCHEMER are strict in all of their arguments; if any one argument fails then the application fails. For example, an expression of the form

`(+` E `(FAIL))`

will fail independent of the expression E. This is the major optimization that our algorithm for dependency analysis performs.

4.5.1 Optimal dependency-analysis is intractable

We mentioned in Section 2 that the smaller a nogood is, the more of the search tree it will prune. Dependency analysis produces a nogood from a failure. Since this nogood will be used to prune the search tree, it is should be as small as possible (subject, of course, to the constraint that a nogood never prunes a solution).

This provides a natural way to evaluate an algorithm for dependency analysis; the smaller the nogoods an algorithm produces, the better that algorithm is. Furthermore, there is an ideal against which to measure nogoods. At a particular failure, there will be at least one subset of the labels above that failure that prunes the largest portion of the search tree without removing any solutions. Such a subset would be an "optimal" nogood. An ideal algorithm for dependency analysis should produce an optimal nogood at each failure. Unfortunately, this is impossible.

Undecidability Theorem: The problem of computing the optimal nogood for a SCHEMER expression is undecidable.

Proof: A SCHEMER expression E fails under all possible executions just in case the empty set $\emptyset$ is a sound nogood for E. The empty set is the smallest possible nogood, so E always fails if and only if $\emptyset$ is the optimal nogood for E. Determining that an arbitrary SCHEMER expression always fails is undecidable; this can be proven in any number of ways, but we will do it by reduction from Post's Correspondence Problem (see [12]).

We can write a SCHEMER procedure `ANY-SEQUENCE` which takes a list of strings as an argument, such that the possible values of `(ANY-SEQUENCE` x`)` are the sequences constructed from the strings of x. Let our input lists of strings be `A` and `B`, and consider the following expression.

```
(LET ((A-SEQUENCE (ANY-SEQUENCE A))
      (B-SEQUENCE (ANY-SEQUENCE B)))
  (IF (EQUAL A-SEQUENCE B-SEQUENCE)
      T
      (FAIL)))
```

The above expression always fails just in case there is no solution to the correspondence problem for the sequences A and B. □

One might doubt the relevance of this theorem, since it relies upon SCHEMER expression with an infinite number of possible executions. However, determining the smallest sound nogood is difficult even for expressions with a finite number of possible executions.

Intractability Theorem: The problem of computing the smallest sound nogood for a SCHEMER expression is $\mathcal{NP}$-hard even if the expression has only a finite number of possible executions.

Proof: The argument is very similar to the above, except that we reduce from SAT (see [11]) and we are restricted to expressions with a finite number of possible executions. Let Ψ be the propositional formula Conjunctive Normal Form whose satisfiability we are trying to determine, and let $\varphi_1, \varphi_2 \ldots \varphi_n$ be the propositional symbols occuring in Ψ. Consider the following expression.

```
(LET ((φ1 (AMB T NIL))
      (φ2 (AMB T NIL))
      ⋮
      (φn (AMB T NIL)))
  (IF Ψ
      T
      (FAIL)))
```

This expression has only a finite number (in fact, 2^n) of possible executions. Furthermore, it always fails just in case Ψ is unsatisfiable. □

There is much more that could be said about the way that dependency analysis is done in SCHEMER. It is clearly possible to produce smaller nogoods by, for example, adding some knowledge of arithmetic; the algorithm for dependency analysis described in section 4.5 will claim that the failure of

```
((LAMBDA (X) (IF (= X X) (FAIL) 23)) (AMB 0 1))
```

depends upon the value of X, which is clearly irrelevant.

Another way to produce smaller nogoods is to use "nogood resolution". Suppose we know that both {AMB-39-L, AMB-40-L} and {AMB-39-L, AMB-40-R} are nogoods;

nogood resolution allows us to deduce that {AMB-39-L} is therefore a nogood. Nogood resolution can be viewed as a way of using propositional inference to make smaller (hence, hopefully, better) nogoods from a set of larger ones.

However, producing better nogoods do not necessarily result in better performance on search problems. Nogood resolution, for example, can consume considerable computational resources without necessarily reducing the search space very much. It is an open question whether nogood resolution is actually worthwhile in practice. It is also simple to add nogood resolution to the dependency analysis algorithm described above. In the concluding section of this paper, we will sketch a way of using an arbitrary algorithm for determining CNF propositional satisfiability with SCHEMER; this includes nogood resolution as a special case.

5. Related Work

SCHEMER can be used to define a search *problem* without making any commitment as to how that problem is to be solved. We have shown that it is possible to construct a search mechanism which applies dependency-directed backtracking to any search problem defined in SCHEMER. An interpreter for SCHEMER which makes use of the mechanism described in Section 4 has been implemented in SCHEME; it is contained in [20].

There are several software systems which have been constructed to provide facilities for dependency directed search. The best known systems include Truth Maintenance Systems (or TMS's) [10], deKleer's consumer architecture [8], and the language AMORD [9]. These systems require the user to explicitly provide labels and logical relations which form the basis of dependency analysis. All of these systems record and use logical implications between data objects which can be viewed as statements of propositional logic. Furthermore, all of these systems require that the user give explicit names to propositions, and explicitly state implication relationships between propositions. These systems therefore do not allow for a definition of the search problem which is independent of the search technique.

The programming langauge PROLOG is similar to SCHEMER in that it allows for concise definitions of search problems. Furthermore, there has been considerable work on selective backtracking within the PROLOG community [4] and, more recently, some work on lateral pruning [17]. Using lateral pruning in finding solutions of a PROLOG search problem is similar to using lateral pruning to find values of SCHEMER expression. The primary problem in lateral pruning is labeling the nodes in the search tree so that a given label appears down many different branches. In both PROLOG and SCHEMER this requires an infinite naming process which is independent of the search.

Alan Bawden describes a simple SCHEMER interpreter that uses a TMS as a backend in [3]. Bawden's interpreter manages to do dependency-directed backtracking without using syntactically named expressions, although we believe that his scheme is quite similar to our own. His interpreter also does some additional

lateral transfer of information that the interpreters presented at the end of this paper do not.

Some earlier work done by David Chapman as part of the planning research described in [5] is closely related to SCHEMER. Chapman attempted to use dependency directed pruning techniques in a non-deterministic LISP with side effects.

Unfortunately, side-effects complicate dependency analysis by making it too difficult to show that a certain choice is irrelevant to a failure. In SCHEMER, dependency analysis can be done incrementally. When the variable `X` is bound to the value of `(FOO)`, all the choices that affect the value of `X` can be collected incrementally in the process of evaluating the body of `FOO`, and no other choice can affect the value of `X`. In the code below, the `AMB` shown is always irrelevant to the value of `X`.

```
(LET ((X (FOO)))
  (LET ((Y (AMB (F) (G))))
    (BAR X Y)))
```

In the presence of side-effects it is hard to prove that the value of `X` does not depend on whether `Y` is `(F)` or `(G)`. This is because `(G)` could side-effect data shared with `X`. This makes it difficult to design a method for dependency analysis which is sound in the presence of side-effects. Sound techniques for dependency analysis in the presence of side effects seem to yield large nogoods that are not useful for pruning.

6. Conclusions and Areas for Future Research

Our main thesis is that non-derministic programming languages like SCHEMER and PROLOG provide a way of defining search problems independent of the search techniques. Furthermore, it is possible to apply sophisticated search techniques such as dependency-directed backtracking to arbitrary search problems defined in this way. It should be possible to apply other search strategies to search problems defined in SCHEMER. We are currently examining ways of applying another sophisticated search technique, rearrangement search, to arbitrary SCHEMER expressions.

One approach to applying sophisticated search strategies to arbitrary SCHEMER programs is to first translate a SCHEMER expression E into a Conjunctive Normal Form formula Ψ in the propositional calculus. This formula should have the property that the possible values of E can be determined from the truth assignments which satisfy Ψ. Any technique for finding satisfying truth assignments of a CNF formula can then applied to the formula Ψ. The rearrangement search ideas described in [21], for example, can be used in finding the satisfying truth assignments of Ψ.

This is actually more complex than it sounds, because the equivalent formula need not be finite; instead, the expression E must be incrementally converted into a formula. Here is an outline of a simple way to accomplish this.

For every named AMB in the expression E, create a proposition φ with the meaning that this AMB returns its first argument; the opposite of this proposition $\neg\varphi$ will mean that this AMB returns its second argument. Let the set of these propositions be Φ. We will maintain a CNF formula Ψ over the propositions in Φ as a list of clauses, initially empty. Note that any complete truth assignment to the propositions in Φ defines a possible execution for E.

We start by picking an arbitrary possible execution of the expression E. Let us assume this results in failure[4] . Dependency analysis at this failure produces a set of assumptions that is a nogood, such as {AMB-0-L, AMB-1-R}. This nogood has an associated formula over Φ, in this case $\varphi_0 \wedge \neg\varphi_1$. We can negate this formula to get a clause, in this case $\neg\varphi_0 \vee \varphi_1$. We then add this clause to Ψ. We will perform this process every time we examine a possible execution of E; essentially, we will record all the nogoods in the CNF formula Ψ.

Now instead of picking an arbitrary possible execution to examine next, we can pick one whose associated truth assignment is consistent with Ψ. If Ψ is unsatisfiable, then the nogoods that have been discovered so far rule out all possible executions, and E always fails. Any execution consistent with Ψ is consistent with all the nogoods that have been discovered so far, and hence might produce a solution.

This technique allows one to hook up an arbitrary method for determining propositional satisfiability to a SCHEMER interpreter. Unless the method employed is quite weak, this will give the effect of nogood resolution (something that the interpreters in the appendix choose not to do). More importantly, it shows that quite a variety of search strategies can be used on search problems defined in this simple, elegant language.

6.1 Acknowledgments

It is difficult to overstate the debt that this research (as well as this author) owes to David McAllester. It could not have been done without him. David Chapman's work on DDL also played a vital role. Gerald Sussman supervised the Master's Thesis that this paper is based upon.

Numerous individuals helped with this research, but Alan Bawden deserves special mention. Phil Agre, Jonathan Rees, Mark Shirley, Jeff Siskind, Daniel Weise, Dan Weld and Brian Williams also contributed useful insights. John Lamping and Joe Weening read and commented on earlier drafts of this paper.

This paper describes research done at the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology, supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-80-C-0505 and N00014–86–K–0180, in part by National Science Foundation grant MCS-8117633, and in part by the IBM Corporation. Ramin Zabih is supported by a fellowship from the Fannie and John Hertz Foundation.

[4] We leave the generalization to the case where there are non-failing possible executions as an easy exercise for the reader

Appendix

This appendix contains code for two SCHEMER interpreters, which use chronological backtracking and dependency-directed backtracking. The code is written in SCHEME. The purpose of these interpreters is completely pedagogical.

If you wish to play with the code contained herein, you can either re-type it in yourself or you can obtain it electronically. As of this writing, these sources are available in the directory "ftp/pub/schemer" on the host "gang-of-four.stanford.edu". You can also write to me directly (as "rdz@score.stanford.edu") if you want to know the current state of SCHEMER.

There are several things included here besides the actual two interpreters of interest, in hopes of making things clearer. First there is a simple SCHEME interpreter using continuation-passing style. There follows a simple SCHEMER interpreter which does not do naming, and which uses chronological backtracking. Examining the differences between these first two interpreters should help the reader understand what I call "continuation-mapping style", which is used extensively in the later interpreters.

In a continuation passing evaluator, `EVAL` takes three arguments: an expression, an environment, and a continuation (a function of one argument). The contract of `EVAL` is to invoke the continuation on the value of the expression in the environment. In SCHEMER, however, an expression has a *set* of values. So rather than passing the value of the expression to the continuation, the job of `EVAL` is to produce the stream of possible values of the expression and to `MAP-STREAM` the continuation over this stream. The resulting stream is then returned.

Preliminaries

The file *trivial.scm* contains various support functions needed for all interpreters. Of particular interest is the variable `*GLOBAL-ENV*` which contains the SCHEME primitives that the various interpreters will support. The absence of procedures like `DISPLAY` shows that the code here is didactic rather than practical.

File: trivial.scm

```
;;; -*- Mode: SCHEME; Syntax: SCHEME; Package: SCHEME -*-
;;; Elementary support for various interpreters.
;;; Trivial data types. "Self documenting".
(define (quotation? obj)
  (and (pair? obj) (eq? (car obj) 'quote)))
(define (primitive? obj)
  (and (pair? obj) (eq? (car obj) 'primitive)))
(define (lambda? obj)
  (and (pair? obj) (eq? (car obj) 'lambda)))
```

```
(define (self-evaluating? exp)
  (or (number? exp) (string? exp) (boolean? exp)))

;;; Short names for list selectors
(define second cadr)

(define third caddr)

(define fourth cadddr)

;;; Closures
(define (closure? obj)
  (and (pair? obj) (eq? (car obj) 'closure)))

(define (make-closure exp env)
  (list 'closure exp env))

(define (closure-body c)
  (third (second c)))

(define (closure-args c)
  (second (second c)))

(define (closure-env c)
  (third c))

;;; Environment support
;;; Standard top-level environment
(define *global-env*
  `((+ (primitive ,+))
    (- (primitive ,-))
    (* (primitive ,*))
    (= (primitive ,=))
    (< (primitive ,<))
    (> (primitive ,>))
    (1+ (primitive ,1+))
    (not (primitive ,not))
    (t t)
    (() ())))

(define (primitive-func prim) (cadr prim))

;;; Unbound variable result
(define *unbound* (list '* 'unbound '*))

(define (no-binding? result)
  (eq? result *unbound*))

;;; Environment selector and constructor
(define (lookup var env)
  (cond ((null? env) *unbound*)
        ((eq? var (car (car env)))
         (cadr (car env)))
```

```
        (else (lookup var (cdr env)))))
(define (bind vars vals env)
  (append (map list vars vals) env))
```

End of trivial.scm

The file *streams.scm* contains some additional stream support. It assumes that your implementation of SCHEME provides the following:

CONS-STREAM, HEAD, TAIL, EMPTY-STREAM? and THE-EMPTY-STREAM

If any of these are missing in your implementation you will have to provide them.

An interesting property of the STREAM-APPEND procedure is that if the arguments in the recursive call are reversed, the resulting procedure appends breadth-first rather than depth-first. This will make the SCHEMER interpreters that use syntactically named expressions traverse the search tree in breadth-first, rather than depth-first, order.

File: streams.scm

```
;;; -*- Mode: SCHEME; Syntax: SCHEME; Package: SCHEME -*-
;;; Some additional stream support in Scheme.  Nothing very complicated,
;;; really.
;;; This assumes the existence of the primitives:
;;; CONS-STREAM, HEAD, TAIL, EMPTY-STREAM?,
;;; and THE-EMPTY-STREAM.
(define (make-singleton x) (cons-stream x the-empty-stream))
(define stream-null? empty-stream?)
(define (stream-append s1 s2)                    ; depth first
  (if (stream-null? s1)
      s2
      (cons-stream (head s1)
                   ;; Reversing these two arguments gives breadth-first
                   (stream-append (tail s1) s2))))
(define (stream-map func stream)
  (if (stream-null? stream)
      stream
      (cons-stream (func (head stream))
                   (stream-map func (tail stream)))))
(define (display-stream stream)
  (define (mapstream func stream)
    (if (not (stream-null? stream))
        (begin (func (head stream))
```

```
              (mapstream func (tail stream)))))
  (display " ")
  (mapstream (lambda (v)
               (begin
                 (write v)
                 (display " ")))
             stream)
  (display ""))
(define (stream-flatten streams-stream)
  (if (stream-null? streams-stream)
      streams-stream
      (stream-append (head streams-stream)
                     (stream-flatten (tail streams-stream)))))
```

End of streams.scm

Simple Interpreters

The file *cps-eval.scm* contains a continuation-passing SCHEME evaluator, while *schemer-eval.scm* contains a chronological backtracking SCHEMER evaluator using continuation-mapping style. A careful comparison of these two evaluators (which are almost identical) is advised for those confused by continuation-mapping.

File: cps-eval.scm

```
;;; -*- Mode: SCHEME; Syntax: SCHEME; Package: SCHEME -*-
;;; Interpreter for Scheme, using continuation-passing style.
;;; Evaluator.  Call CONT on the value of EXP in ENV.
(define (scheme-eval exp env cont)
  (cond ((self-evaluating? exp)
         (cont exp))
        ((symbol? exp)                              ; variable
         (let ((val (lookup exp env)))
           (if (no-binding? val)
               (error "Unbound variable" exp)
               (cont val))))
        ((not (pair? exp))
         (error "Unknown expression" exp))
        (else
         (case (car exp)
           ((quote)
            (cont (cadr exp)))
           ((lambda)
```

```
        (cont (make-closure exp env)))
       ((if)
        (let ((predicate (cadr exp))
              (cons (caddr exp))
              (alt (cadddr exp)))
          (scheme-eval predicate env
                       (lambda (pred)
                         (if pred
                             (scheme-eval cons env cont)
                             (scheme-eval alt env cont))))))
       (else
         (let ((args (cdr exp))
               (func (car exp)))
           (scheme-eval func env
                        (lambda (function)
                          (eval-args args env '()
                                     (lambda (evaled-args)
                                       (scheme-apply
                                         function
                                         evaled-args
                                         cont)))))))))
```

```
;;; Call CONT on the list of values of the elements of ARGS
;;; in ENV.
(define (eval-args args env answer cont)
  (if (null? args)
      (cont (reverse answer))
      (scheme-eval (car args) env
                   (lambda (ans)
                     (eval-args (cdr args) env
                                (cons ans answer) cont)))))
;;; Call CONT on the result of applying FUNC to ARGS.
(define (scheme-apply func args cont)
  (cond ((eq? (car func) 'primitive)
         (cont (apply (cadr func) args)))
        ((closure? func)
         (scheme-eval (closure-body func)
                      (bind (closure-args func) args
                            (closure-env func))
                      cont))
        (else (error "Invalid function to apply" func))))
;;; Interpreter loop
(define (mini-scheme)
  (display "SCHEME==> ")
```

```
(scheme-eval (read)
             *global-env*
             (lambda (value)
               (write value)
               (newline)
               (mini-scheme))))
```

End of cps-eval.scm

File: schemer-eval.scm

```
;;; -*- Mode: SCHEME; Syntax: SCHEME; Package: SCHEME -*-
;;; Interpreter for SCHEMER, using a variant of continuation-passing
;;; style.  This is probably the simplest SCHEMER interpreter I ever
;;; wrote, but it's still somewhat complex.  It uses chronological
;;; backtracking.

;;; The evaluator.  Produce the flattened stream of values that result
;;; from calling CONT on each possible value of EXP in ENV.  This value
;;; is *returned*.
(define (schemer-eval exp env cont)
  (cond ((self-evaluating? exp)
         (cont exp))
        ((symbol? exp)                          ; variable
         (let ((val (lookup exp env)))
           (if (no-binding? val)
               (error "Unbound variable" exp)
               (cont val))))
        ((not (pair? exp))
         (error "Unknown expression" exp))
        (else
         (case (car exp)
           ((quote)
            (cont (cadr exp)))
           ((lambda)
            (cont (make-closure exp env)))
           ((if)
            (let ((predicate (cadr exp))
                  (cons (caddr exp))
                  (alt (cadddr exp)))
              (schemer-eval predicate env
                            (lambda (pred)
                              (if pred
                                  (schemer-eval cons env cont)
```

```
                                (schemer-eval alt env cont))))))
            ;; Return a stream of values
            ((amb)
             (stream-append (schemer-eval (second exp) env cont)
                            (schemer-eval (third exp) env cont)))
            ;; Return the empty stream
            ((fail)
             the-empty-stream)
            (else
              (let ((args (cdr exp))
                    (func (car exp)))
                (schemer-eval func env
                              (lambda (function)
                                (schemer-eval-args args env '()
                                                   (lambda (vals)
                                                     (schemer-apply function vals
                                                                    cont)))))))))))
;;; Return the stream of values created by calling CONT on
;;; the values of the elements of ARGS in ENV.
(define (schemer-eval-args args env answer cont)
  (if (null? args)
      (cont (reverse answer))
      (schemer-eval (car args) env
                    (lambda (ans)
                      (schemer-eval-args (cdr args) env (cons ans answer)
                                         cont)))))
;;; Do the application of FUNC to ARGS, calling CONT on the result.
(define (schemer-apply func args cont)
  (cond ((eq? (car func) 'primitive)
         (cont (apply (cadr func) args)))
        ((closure? func)
         (schemer-eval (closure-body func)
                       (bind (closure-args func) args
                             (closure-env func))
                       cont))
        (else (error "Invalid function to apply" func))))

;;; Interpreter loop
(define (schemer)
  (newline)
  (display "SCHEMER==> ")
  (let ((exp (read)))
    (display-stream
      (schemer-eval exp
```

```
                    *global-env*
                    (lambda (value) (make-singleton value))))
    (schemer)))
```

End of schemer-eval.scm

Naming interpreters

Now we come to the most important toy interpreters, the ones that use the algorithms described in the body of this paper. First we have the implementation of the naming algorithm itself, contained in the file *namer.scm*. This is surprisingly complex. Much of the complexity is due to the use of lazy S-expressions in order to cause delayed evaluation to work correctly. In addition, there is a reasonable amount of simple code to support tagged datastructures.

An important note is that the "memoization" of syntactically named expressions is *not* merely an efficiency optimization; instead, it is required to get dependency-directed backtracking to work correctly. Because the `AMB` counter is kept in a global variable, not memoizing a syntactically named expression could result in the identity of a choice changing over time.

File: namer.scm

```
;;; -*- Mode: SCHEME; Syntax: SCHEME; Package: SCHEME -*-
;;; SCHEMER Namer. This takes a SCHEMER expression and turns it into a
;;; named choice expression. Since the result may be infinite, we use
;;; delayed evaluation all over the place.
(define (name-eval exp env)
  (cond ((named? exp)
         exp)
        ((self-evaluating? exp)
         (make-named-constant exp))
        ((symbol? exp)
         (let ((val (lookup exp env)))
           (if (no-binding? val)
               (error "Unbound variable" exp)
               val)))
        ((not (pair? exp))
         (error "Invalid expression" exp))
        ((eq? (car exp) 'quote)
         (cadr exp))
        (else
         (case (car exp)
           ((lambda) (make-named-closure exp env))
```

```
        ((if)
          (let ((pred (name-eval (second exp) env)))
            (if (constant? pred)
                (if (test pred)
                    (name-eval (third exp) env)
                    (name-eval (fourth exp) env))
                (make-named-conditional pred
                      (delay (name-eval (third exp) env))
                      (delay (name-eval (fourth exp) env))))))
        ((amb) (make-named-choice (new-choice)
                     (delay (name-eval (second exp) env))
                     (delay (name-eval (third exp) env))))
        ((fail) failure)
        (else
          (name-apply (name-eval (car exp) env)
                      (map (lambda (arg) (name-eval arg env))
                           (cdr exp)))))))

(define (name-apply func args)
  (cond ((and (constant? func) (primitive? (constant-value func)))
         (cond ((some (lambda (arg) (failure? arg)) args)
                failure)
               ((every constant? args)
                (make-named-constant
                  (apply (primitive-func (constant-value func))
                         (map constant-value args))))
               (else
                (make-named-application func args))))
        ((named-closure? func)
         (let* ((lambda (named-closure-exp func))
                (env (named-closure-env func))
                (body (third lambda))
                (lvars (second lambda)))
           (name-eval body (bind lvars args env))))
        ((named-choice? func)
         (let* ((left-function (choice-left func))
                (right-function (choice-right func))
                (number (choice-number func))
                (named-args (map name-eval args)))
           (make-named-choice number
                      (delay (name-apply left-function named-args))
                      (delay (name-apply right-function named-args)))))
        ((named-conditional? func)
         (let ((pred (conditional-test func)))
```

```
            (if (constant? pred)                    ;; if constant, do it now
                (if (test pred)
                    (name-apply (conditional-conseq func) args)
                    (name-apply (conditional-alter func) args))
                (make-named-conditional                    ;; otherwise delay
                     pred
                     (delay (name-apply (conditional-conseq func) args))
                     (delay (name-apply (conditional-alter func) args)))
                     )))
          ((named-application? func)
           (error "This primitive does not return an applicable function"
                  (constant-value func)))
          (else (error "Unknown function type" func))))

;;; All the code below is datastructure support for named choice
;;; expressions.
;;; Support for thunks
(define thunk-tag (list '* 'thunk-tag '*))

(define (make-thunk promise)
  ;; promise must be the result of (DELAY exp)
  (cons thunk-tag promise))

(define (thunk? thing)
  (and (pair? thing)
       (eq? (car thing) thunk-tag)))

(define (force-thunk thunk)
  (force (cdr thunk)))

;;; Mini-structure support
(define (tag? thing val)
  (and (pair? thing) (eq? (car thing) val)))

(define closure-tag (list '* 'closure-tag '*))
(define application-tag (list '* 'application-tag '*))
(define conditional-tag (list '* 'conditional-tag '*))
(define choice-tag (list '* 'choice-tag '*))
(define failure-tag (list '* 'failure '*))
(define constant-tag (list '* 'constant-tag '*))

(define (named? thing)
  (and (pair? thing)
       (memq (car thing) name-tags)))

(define name-tags
  (list closure-tag application-tag conditional-tag choice-tag
        failure-tag constant-tag))

;;; Constants, closures and failures come first because they are not
```

```
;;; delayed, and are therefore easier to deal with.
;;; Constants
(define (constant? thing)
  (tag? thing constant-tag))
(define (make-named-constant val)
  (list constant-tag val))
(define (constant-value thing)
  (second thing))

;;; Closures
(define (named-closure? thing)
  (tag? thing closure-tag))
(define (make-named-closure exp env)
  (list closure-tag exp env))
(define (named-closure-exp thing)
  (second thing))
(define (named-closure-env thing)
  (third thing))

;;; Failures
(define (failure? thing)
 (tag? thing failure-tag))
(define failure (list failure-tag))

;;; Primitive applications, it turns out, also do not need to be cached.
(define (named-application? thing)
  (tag? thing application-tag))
(define (make-named-application prim args)
  (list application-tag prim args))
(define (application-prim thing)
  (second thing))
(define (application-args thing)
  (third thing))

;;; Conditionals and choices are cached and thus more complex.
;;; Conditionals
(define (named-conditional? thing)
  (tag? thing conditional-tag))
(define (make-named-conditional pred conseq alt)
  (list conditional-tag
        pred
        (make-thunk conseq)
        (make-thunk alt)))
```

```
(define (conditional-test thing)
  (second thing))

(define (conditional-conseq thing)
  (let ((test (third thing)))
    (if (thunk? test)
        (let ((result (force-thunk test)))
          (set-car! (cddr thing) result)
          result)
        test)))

(define (conditional-alter thing)
  (let ((test (fourth thing)))
    (if (thunk? test)
        (let ((result (force-thunk test)))
          (set-car! (cdddr thing) result)
          result)
        test)))

;;; Choices
(define (named-choice? thing) (tag? thing choice-tag))

(define (make-named-choice number left right)
  (list choice-tag number
        (make-thunk left) (make-thunk right)))

(define choice-counter 0)

(define (new-choice)
  (set! choice-counter (1+ choice-counter))
  choice-counter)

(define (choice-number choice) (cadr choice))

(define (choice-left thing)
  (let ((test (third thing)))
    (if (thunk? test)
        (let ((result (force-thunk test)))
          (set-car! (cddr thing) result)
          result)
        test)))

(define (choice-right thing)
  (let ((test (fourth thing)))
    (if (thunk? test)
        (let ((result (force-thunk test)))
          (set-car! (cdddr thing) result)
          result)
        test)))

;;; Miscellaneous utilities
```

```
(define (test node)
  (not (and (constant? node) (eq? (constant-value node) '()))))
(define (terminal? node) (or (constant? node) (failure? node)))
(define *namer-global-env*
  (map (lambda (pair)
         (list (car pair) (make-named-constant (cadr pair))))
       *global-env*))
```

End of namer.scm

Next we have the simplest SCHEMER interpreter that uses syntactically named expressions, *solver-chron.scm*. (For historical reasons best forgotten, the programs that take a syntactically named expression and returned its possible values were at one point called "solvers".) New assumptions are made and added to the variable CONTEXT as the search progresses. This interpreter uses depth-first chronological backtracking (as mentioned, with a small change to STREAM-APPEND it can be made to do breadth-first chronological backtracking). The top-level continuation is included in the very last file of this Appendix.

File: solver-chron.scm

```
;;; -*- Mode: SCHEME; Syntax: SCHEME; Package: SCHEME -*-
;;; Simplest possible way of determining the value of a named choice
;;; expression, using chronological backtracking.  NODE is a named
;;; choice expression, CONTEXT is a set of assumptions.
;;; CONT is a continuation taking a new named choice expression and
;;; a new context (a superset of the old one).
(define (chron-eval node context cont)
  (cond ((terminal? node)
         (cont node context))
        ((named-conditional? node)
         (let ((pred (conditional-test node))
               (conseq (conditional-conseq node))
               (alter (conditional-alter node)))
           (chron-eval pred context
                       (lambda (pred-val new-context)
                         (cond ((failure? pred-val)
                                (cont pred-val new-context))
                               ((constant-value pred-val)
                                (chron-eval conseq new-context cont))
                               (else
                                (chron-eval alter new-context cont)))
                         ))))
```

```
        ((named-choice? node)
         (let ((probe (assumption node context)))
           (if (empty? probe)
               (let ((left (choice-left node))
                     (right (choice-right node)))
                 (stream-append
                   (chron-eval left
                               (add-assumption node left context)
                               cont)
                   (chron-eval right
                               (add-assumption node right context)
                               cont)))
               (chron-eval probe context cont))))
        ((named-application? node)
         (let ((func (primitive-func
                       (constant-value (application-prim node))))
               (args (application-args node)))
           (chron-eval-args args '() context
                            (lambda (values new-context)
                              (cont (if (failure? values)
                                        values
                                        (make-named-constant
                                          (apply func
                                                 (map constant-value
                                                      values))))
                                    new-context)))))
        (else (error "Unknown node to CHRON-EVAL" node))))

(define (chron-eval-args args values context cont)
  (if (null? args)
      (cont (reverse values) context)
      (chron-eval (car args) context
                  (lambda (value new-context)
                    (if (failure? value)
                        (cont value context)
                        (chron-eval-args (cdr args) (cons value values)
                                         new-context cont))))))
```

End of solver-chron.scm

Dependency-directed backtracking in SCHEMER

Finally, we have the payoff: a SCHEMER interpreter that uses dependency-directed backtracking. First we need some support for storing nogoods and checking

if a given context is a (not necessarily proper) superset of a nogood, and hence inconsistent. In any real implementation, this would be done using a TMS (readers interested in how this might work are urged to read Alan Bawden's interpreter described in [3]). In this toy interpreter, a set of nogoods is represented as a list, and new nogoods are simply added to the front. The file *contexts.scm* contains support for the nogood database.

File: contexts.scm

```
;;; -*- Mode: SCHEME; Syntax: SCHEME; Package: SCHEME -*-
;;; Support for assumptions, contexts and justifications. Includes the
;;; inconsistency cache (the current set of nogoods).
;;; These two utility functions behave as in COMMON-LISP.
(define (every pred list)
  (or (null? list)
      (and (pred (car list))
           (every pred (cdr list)))))
(define (some pred list)
  (and list
       (or (pred (car list))
           (some pred (cdr list)))))
;;; Context support. An assumption is a pair of a named choice and a
;;; value. A context is a list of assumptions.
;;; The empty context
(define empty-context '())
(define no-value `(* no-value *))
;;; Lookup the value of the assumption about this node in this context
(define (assumption node context)
  (let ((entry (assq node context)))
    (if entry (cdr entry) no-value)))
;;; Is this assumption empty?
(define (empty? assumption)
  (eq? assumption no-value))
;;; Add the assumption that this choice has this value
(define (add-assumption choice value context)
  (cons (cons choice value) context))
;;; Justifications are also contexts
;;; Empty justification
(define empty-just empty-context)
;;; Union 2 justifications together
(define (union-just j1 j2)
  (append j1 j2))
```

```
;;; Is context an extension (i.e. superset) of this nogood
(define (extension-of? context nogood)
  (every (lambda (assumption)
           (member assumption context))
         nogood))

;;; Nogood support. This is absurdly inefficient. A real
;;; implementation would use a TMS. Instead, we keep a list of nogoods
;;; (contexts known to be inconsistent).
;;; The inconsistent contexts
(define *inconsistent-contexts* '())

;;; Is this context inconsistent?
(define (inconsistent? context)
  (some (lambda (nogood)
          (extension-of? context nogood))
        *inconsistent-contexts*))

;;; Make this context inconsistent!
(define (nogood! context)
  (set! *inconsistent-contexts*
        (cons context *inconsistent-contexts*)))
```

End of contexts.scm

The final SCHEMER is contained in *solver-ddb.scm*. New assumptions are collected in CONTEXT, while the assumptions that will form a nogood if the current expression fails are passed along to the continuation. The first thing done is to check if the input context has become inconsistent; if so, a message is printed and backtracking occurs.

File: solver-ddb.scm

```
;;; -*- Mode: SCHEME; Syntax: SCHEME; Package: SCHEME -*-
;;; Determine the value of the named choice expression NODE using
;;; dependency-directed backtracking. CONTEXT is the context in which
;;; we are to determine the possible values of NODE. CONT is a
;;; continuation taking a new named choice expression, a new context (a
;;; superset of CONTEXT), and a justification for the fact that NODE has
;;; this value in CONTEXT.
(define (ddb-eval node context cont)
  (cond ;; Check for inconsistency
        ((inconsistent? context)
         ;; Tell the user it's done something.
         (display "Pruning search tree...")
```

```
 the-empty-stream)
((terminal? node)
 (cont node context empty-just))
((named-conditional? node)
 (let ((pred (conditional-test node))
       (conseq (conditional-conseq node))
       (alter (conditional-alter node)))
   (ddb-eval
     pred context
     (lambda (pred-val pred-context pred-just)
       (let ((new-cont
              (lambda (case-val case-context case-just)
                (cont case-val case-context
                      (union-just pred-just case-just)))))
         (cond ((failure? pred-val)
                (cont pred-val pred-context pred-just))
               ((constant-value pred-val)
                (ddb-eval conseq pred-context new-cont))
               (else
                (ddb-eval alter pred-context new-cont)))))))))
((named-choice? node)
 (let* ((probe-proc (assumption node context))
        (left-context (add-assumption node choice-left context))
        (right-context (add-assumption node choice-right context))
        (left-cont (lambda (value new-context new-just)
                     (cont value new-context
                           (add-assumption node choice-left
                                           new-just)
                           )))
        (right-cont (lambda (value new-context new-just)
                      (cont value new-context
                            (add-assumption node choice-right
                                            new-just)
                            ))))
   (if (empty? probe-proc)
       (stream-append
         (ddb-eval (choice-left node) left-context left-cont)
         (ddb-eval (choice-right node) right-context right-cont))
       (ddb-eval (probe-proc node)
                 context
                 (lambda (value new-context just)
                   (cont value new-context
                         (add-assumption node probe-proc
                                         just)))))))
```

```
        ((named-application? node)
         (let ((func (primitive-func (constant-value
                                      (application-prim node))))
               (args (application-args node)))
           (ddb-eval-args args '() '() context
                          (lambda (arg-values arg-context arg-just)
                            (cont (if (failure? arg-values)
                                      arg-values
                                      (make-named-constant
                                       (apply func
                                              (map constant-value
                                                   arg-values))))
                                  arg-context arg-just)))))
        (else (error "Unknown node to DDB-EVAL" node))))
(define (ddb-eval-args args just values context cont)
  (if (null? args)
      (cont (reverse values) context just)
      (ddb-eval (car args) context
                (lambda (value value-context value-just)
                  (if (failure? value)
                      (cont value value-context value-just)
                      (ddb-eval-args
                        (cdr args)
                        (union-just value-just just)
                        (cons value values)
                        value-context cont))))))
```

End of solver-ddb.scm

The top-level loops for the two important interpreters are contained in *schemer.scm*. One can obtain a chronologically backtracking intereter by calling the procedure `CHRONOLOGICAL-SCHEMER`, and an interpreter that uses dependency-directed backtracking by calling `DDB-SCHEMER`.

File: schemer.scm

```
;;; -*- Mode: SCHEME; Syntax: SCHEME; Package: SCHEME -*-
;;; Call this function to get a chronologically backtracking SCHEMER
;;; interpreter.
(define (chronological-schemer)
  (schemer-loop
    (lambda (named-choice-exp)
      (chron-eval named-choice-exp empty-context
```

```
                    (lambda (terminal-node context)
                      (if (failure? terminal-node) the-empty-stream
                          (make-singleton terminal-node)))))))

;;; Call this function to get an interpreter with DDB.
(define (ddb-schemer)
  (schemer-loop
    (lambda (named-choice-exp)
      ;; Forget all previous nogoods
      (set! *inconsistent-contexts* '())
      (ddb-eval named-choice-exp empty-context
                (lambda (terminal-node context justification)
                  (cond ((failure? terminal-node)
                         ;; Record the inconsistency
                         (nogood! justification)
                         ;; Return empty stream
                         the-empty-stream)
                        (else
                         (make-singleton terminal-node)))))))))

;;; The basic driver loop.  SOLVER simply turns a named choice
;;; expression into a stream of values (ie. terminal nodes).
(define (schemer-loop solver)
  (display "SCHEMER==> ")
  (let ((exp (read)))
    (display-stream
      (stream-map (lambda (node)
                    (cond ((constant? node)
                           (constant-value node))
                          ((named-closure? node)
                           "#[Closure]")
                          (else
                           (error
                            "This value is not a valid terminal node"
                            node))))
                  (solver (name-eval exp *namer-global-env*))))
    (newline)
    (schemer-loop solver)))
```

End of schemer.scm

The anonymous continuation used in `DDB-SCHEMER` is particularly important, as it takes a nogood and records it in the database. The simplest example where dependency-directed backtracking will occur is something like:

```
((LAMBDA (X)
   (+ X ((LAMBDA (Y)
           (IF (= Y 0) (FAIL) Y))
         (AMB 0 1))))
 (AMB 20 30))
```

The choice of a value for `X` is irrelevant to the failure. Let us name the choices `(AMB-0 0 1)` and `(AMB-1 20 30)`. The first (leftmost) failure will occur with the context {`AMB-0-L`, `AMB-1-L`}. The nogood that dependency analysis will produce from this failure will consist only of {`AMB-0-L`}. As a result the context {`AMB-0-L`, `AMB-1-R`} will be pruned by dependency-directed backtracking.

References

1. Abelson, H. and Sussman, G., with Sussman, J., *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

2. Barendregt, H., *The Lambda Calculus*, Studies in Logic and the Foundations of Mathematics, Volume 103, North-Holland, 1984.

3. Bawden, A., A normal order evaluator for nondeterministic Lisp, MIT AI Working Paper 304, Cambridge, MA, 1987.

4. Bruynooghe, M. and Pereira, L., Deduction revision by intelligent backtracking, in: *Implementations of Prolog*, J. Campbell (Ed.), Ellis Horwood, 1984.

5. Chapman, D., Planning for conjunctive goals, MIT AI Technical Report 802, November 1985. Revised version appears in: *Artificial Intelligence* **32** (1987), 333–377.

6. Clinger, W., Nondeterministic call by need is neither lazy nor by name, in: *Proceedings of the ACM Conference on LISP and Functional Programming*, 226–234, 1982.

7. Cohen, J., Non-deterministic algorithms, *Computing Surveys* **11** (1979), 79–94.

8. deKleer, J., Problem solving with the ATMS, *Artificial Intelligence* **28** (1986), 197–224.

9. deKleer, J., Doyle, J., Steele, G. and Sussman, G., Explicit control of reasoning, in: *Artificial Intelligence: an MIT Perspective*, P. Winston and R. Brown (Eds.), MIT Press, Cambridge, MA, 1982.

10. Doyle, J., A truth maintenance system, *Artificial Intelligence* **12** (1979), 231–272.

11. Garey, M. and Johnson, D., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, (Freeman, New York, 1979).

12. Hopcroft, J. and Ullman, J., *Introduction to Automata Theory, Languages, and Computation*, (Addison-Wesley, 1979).

13. McCarthy, J., A basis for a mathematical theory of computation, in: *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.), North-Holland, Amsterdam, 1963.

14. Montanari, U., Networks of constraints: fundamental properties and applications to picture processing, *Information Sciences* **7** (1974) 95–132.

15. Purdom, P., Brown, C. and Robertson, E., Multi-level dynamic search rearrangement, *Acta Informatica* **15** (1981) 99–114.

16. Rees, J. *et. al.*, Revised^3 report on the algorithmic language Scheme, ACM SIGPLAN Notices **21** (12), December 1986.

17. Siskind, J., personal communication.

18. Stallman, R. and Sussman, G., Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis, *Artificial Intelligence* **9** (1977), 135–196.

19. Warren, D., Pereira, L. and Pereira, F., Prolog — the language and its implementation compared with Lisp, in: *ACM Symposium on Artificial Intelligence and Programming Languages*, 1977.

20. Zabih, R., Dependency-directed backtracking in non-deterministic Scheme, M.S. thesis, MIT Department of Electrical Engineering and Computer Science, January 1987.

21. Zabih, R. and McAllester, D., A rearrangement search strategy for determining propositional satisfiability, in: *Proceedings AAAI-88*, St. Paul, MN (1988).

22. Zabih, R., McAllester, D. and Chapman, D., Non-deterministic Lisp with dependency-directed backtracking, in: *Proceedings AAAI-87*, Seattle, WA, (1987) 59–64.