Non-Deterministic Lisp with Dependency-Directed Backtracking

Ramin Zabih[†], **David McAllester** and **David Chapman**
Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Abstract

Extending functional Lisp with McCarthy's non-deterministic operator AMB yields a language which can concisely express search problems. Dependency-directed backtracking is a powerful search strategy. We describe a non-deterministic Lisp dialect called SCHEMER and show that it can provide automatic dependency-directed backtracking. The resulting language provides a convenient interface to this efficient backtracking strategy.

Many problems in Artificial Intelligence involve search. SCHEMER is a Lisp-like language with non-determinism which provides a natural way to express search problems. Dependency-directed backtracking is a powerful strategy for solving search problems. We describe how to use dependency-directed backtracking to interpret SCHEMER. This provides SCHEMER programs with the benefits of dependency-directed backtracking automatically.

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. Finally, we argue that SCHEMER with automatic dependency-directed backtracking would be a useful tool for Artificial Intelligence by comparing it with current methods for obtaining dependency-directed backtracking.

I. SCHEMER is Scheme with AMB

SCHEMER consists of functional Scheme [Rees *et al.* 1986] plus McCarthy's ambiguous operator AMB [McCarthy 1963] and the special form (FAIL). AMB takes two arguments and non-deterministically returns the value of one of them. Selecting the arguments of the AMB's in an expression determines a possible execution. Each SCHEMER expression is thus associated with a set of possible values. In the program below, the expression (ANY-NUMBER) non-deterministically returns some whole number.

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

Similarly, (ANY-PRIME) non-deterministically returns some prime number.

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

ANY-PRIME eliminates certain possible values by evaluating (FAIL). The expression (FAIL) has no possible values.

A mathematically precise semantics for SCHEMER is beyond the scope of this paper — there are several possible semantics that differ in technical detail [Clinger 1982, Zabih *et al.* 1987]. Under all these semantics, however, the expression (FAIL) can be used to eliminate possible values; finding a possible value for a SCHEMER expression requires finding an execution that doesn't evaluate (FAIL).

For a given expression there may be a very large number of different ways of choosing the values of AMB expressions. If there are n independent binary choices in the computation then there are 2^n different combinations of choices, and thus 2^n different executions. In certain expressions most combinations of choices result in failure. Finding one or more possible values for a SCHEMER expression requires searching the various possible combinations of choices.

Interpreting SCHEMER thus requires search. The semantics of the language do not specify a search strategy. Correct interpreters with different strategies will produce the same possible values for an expression, and can differ only in efficiency. It is straightforward to write a SCHEMER interpreter that searches all possible executions in a brute force manner by backtracking to the most recent non-exhausted choice in the event of a failure. Such an interpreter would use simple "chronological" backtracking.

We describe a more sophisticated SCHEMER interpreter that automatically incorporates dependency analysis and dependency-directed backtracking. This inter-

---

[†] Author's current address: Computer Science Department, Stanford University, Stanford, California, 94305.

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.

Figure 1: A search tree. Failures are labeled "f".

preter, originally described in [Zabih 1987], allows programmers to gain the efficiency benefits of dependency-directed backtracking automatically for SCHEMER code.

II. Dependency-Directed Backtracking

Dependency-directed backtracking is a general search strategy invented by Stallman and Sussman [Stallman and Sussman 1977]. It can best be understood as a technique for pruning search trees. Consider an arbitrary search tree generated by some particular search. Such a tree is shown in Figure 1. The leaves of the tree labeled with the letter "f" represent search paths which lead to failure. Dependency-directed backtracking can be used to prune such a search tree, by detecting unsearched fragments of the tree which cannot contain solutions.

Dependency-directed backtracking requires that two additional pieces of information be added to the tree. First, the non-root nodes must be assigned labels. Second, each failing leaf node must be associated with a subset of the set of labels that appear above that leaf. For reasons to be explained, the process of assigning sets of labels to failing leaf nodes is called *dependency analysis*. Carrying out labeling and dependency analysis on the tree of Figure 1 could result in Figure 2.

Each label represents a statement that is known to be true of all leaf nodes beneath the labeled node. For example, suppose 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.

Figure 2: The search tree after labeling and dependency analysis. Capital letters are labels. The dependency set for the leftmost failure is also shown.

In this case the label A might represent the statement that n is assigned the color red. All candidate colorings under the search node labeled A would color n red.

The leftmost leaf node in the tree of Figure 2 has been assigned the *dependency set* $\{C,E\}$. 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. 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. Such a set of labels is called a *nogood*.

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 entire search tree. If the nogood had contained the single label C, about half of the tree would have been pruned by this one nogood. In general, the smaller the number of labels in a nogood, the larger the fragment of the search tree pruned by that nogood.

More formally, let N be a nogood, i.e. a set of labels. We say that N *prunes* a given node if every label in N appears above that leaf node in the search tree. 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* if the nogoods associated with failure nodes only prune failure nodes; solution nodes should never be pruned. When computation is required to determine failure, dependencies must be maintained in a way that ensures soundness. If a label contributes to a failure, but the contribution is overlooked, solutions can be missed. For example, if dependency analysis on the leftmost failure in Figure 2 overlooked the contribution of C, a nogood consisting of just $\{E\}$ would be created, which would discard the only solution.

The next two sections describe techniques for automatic node labeling and dependency analysis in SCHEMER. The automatic dependency-directed backtracking provided by these techniques makes it possible for programmers to take advantage of dependency-directed tree-pruning without the necessity of writing their own code for search node labeling and dependency analysis.

III. Node Labeling

Finding one or more of the possible values for a given SCHEMER expression involves searching the possible executions for one which does not require evaluating `(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.

Recall that a label on a node in a search tree represents a statement that is true of all candidate solutions under that node. In SCHEMER search trees the labels represent statements of the form "`AMB-37` chooses its first argument" where `AMB-37` refers to a particular `AMB` expression. For this to work properly, we need to identify particular `AMB` expressions within a given SCHEMER expression; each `AMB` expression must be given a unique name.

Figure 3 shows an expression in which each `AMB` has been given a unique name. The corresponding search tree is also shown. The non-root nodes 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 `AMB-39` chose its first argument. The nogood consisting of the single label `AMB-39-L` prunes the first, third and fifth leaf nodes.

The choices in a SCHEMER expression must be named before searching for possible values. If the naming is done during the search process, there is danger of giving the same `AMB` expression different names in different regions of the search tree. This problem can be avoided

```
(LET ((X (AMB-37 3 (AMB-38 4 5)))
      (Y (AMB-39 6 7)))
  (IF (= Y 6) (FAIL) (+ X Y)))
```

Figure 3: A SCHEMER expression with named choices and its labeled search tree.

by naming all the choices in the expression before starting the search process.

Unfortunately this is not as easy as it might seem. For example consider the expression `(ANY-NUMBER)` defined previously.

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

The above `AMB` expression cannot simply be identified as, say, `AMB-52`, because it is being used to make several different choices in different recursive calls to the procedure `ANY-NUMBER`.

It is possible, however, to "unwind" the recursive calls in the above expression and then to name each choice independently. The resulting expression is called a *named choice expression*. The following infinite named choice expression is the result of unrolling the above definition.

```
(AMB-52 0
  (1+ (AMB-53 0
        (1+ (AMB-54 0
              (1+ ...))))))
```

In the above expression each distinct choice has been given a distinct name. Infinite expressions such as this one can be represented by lazy S-expressions. Lazy S-expressions are analogous to streams [Abelson and Sussman 1985]; lazy

S-expressions 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.

To find the possible values of a SCHEMER expression, the expression is first converted to a named choice expression by giving all AMB expressions names. In practice the result is a lazy S-expression whose parts are computed on demand and then saved. Conceptually, however, the entire named choice expression is created, and all choices are named, before the search process begins. The search process then evaluates the resulting named choice expression. Nodes in the search tree are given labels of the form AMB-52-L which means that AMB expression 52 chooses its first (left) argument.

Producing a named choice expression from a regular SCHEMER expression turns out to be difficult. β-substitution followed by textual naming of AMB's is sufficient for the examples we have mentioned, but does not preserve the semantics of SCHEMER. This is because substitution can result in multiple choices where there should be only one. Consider the procedure below.

```
(DEFINE (BETA)
  ((LAMBDA (X) (+ X X)) (AMB 1 2)))
```

The possible values of (BETA) should be 2 and 4. Performing β-substitution produces an expression with possible values 2, 3 and 4.

It turns out that it is possible to unwind a SCHEMER expression completely so that the resulting named choice expression has the same possible values as the original expression. The basic trick is to interleave β-substitution and textual choice-naming. However, there are several subtleties involved, and the solution is too complex to describe in the space available. Interested readers are referred to [Zabih *et al.* 1987], which contains a complete description of the problem and its solution. Unwinding SCHEMER expressions without violating the semantics of the language was the major technical contribution of [Zabih 1987]. For our present purposes it is only important that a solution exists.

IV. Dependency Analysis

Since SCHEMER expressions can be converted to named choice expressions, the problem of finding possible values for SCHEMER expressions is reduced to the problem of finding possible values for named choice expressions. It is possible to give a simple recursive definition for named choice expressions.

A named choice expression is one of the following, where E_i's denote named choice expressions.

- A constant
- Failure
- A named AMB expression of the form (AMB-n E_l E_r)
- A conditional (IF E_{pred} E_{conseq} E_{alter})
- A primitive application (P E_1 E_2), where P is a Scheme primitive such as +

A given named choice such as AMB-52 may appear in several different places in a given named choice expression. We require that when this happens the arguments to the AMB-52 are the same in all cases. Named choice expressions need not be finite; they are produced top down in a lazy manner.

A set of assumptions about the choices in a named choice expression E will assign E a value. The value is computed by replacing all the named choices by their first or second arguments, depending upon the assumption about that choice. The resulting expression contains no choices at all, and either fails or has a unique possible value.

As the search for possible values of a named choice expression proceeds, assumptions are made about the various choices in the expression. When a value for an expression (or subexpression) is found *dependency analysis* is performed to determine the assumptions about choices which lead to this particular value.

Recall that the job of dependency analysis is to provide a set of labels that constitute a nogood. In SCHEMER, the labels are assumptions such as AMB-57-L. A *justification* for a value of a named choice expression is a set of such assumptions which ensures that the expression has that value. A justification for the value of failure will therefore be a valid nogood.

The justification for a value of a named choice expression can be defined recursively in terms of the justifications for its subexpressions.

- If the expression is a constant or failure, the justification for its value is empty.
- If the expression is a choice (AMB-n E_l E_r), then the justification for its value is the assumption AMB-n-L or AMB-n-R, added to the justification for the value of E_l or E_r, respectively.
- If the predicate of a conditional expression fails, then the entire conditional fails, and the justification for this failure is equal to the justification for the failure of the predicate. If the predicate does not fail then the justification for the value of the conditional is the the union of the justification for the value of the predicate and the justification for the value of whichever branch is taken.
- If any argument to a primitive application fails then the application itself fails, and the justification for this failure equals the justification for the failure of the argument. If no argument fails, the justification for the value of the application is the union of the justifications for the arguments.

Justifications are calculated incrementally as the search progresses. When the search produces a leaf node, which

is a value for the named choice expression, a justification for that value is also produced. If the value is failure, then the justification will be recorded as a nogood.

The search process maintains a list of nogoods, initially empty. Whenever the search discovers a failure, dependency analysis produces a nogood, i.e. a set of assumptions that ensures that the named choice expression fails. This new nogood is added to the list. The search process discards portions of the search tree that are pruned by any of the nogoods.

Automatic dependency-directed backtracking in the SCHEMER interpreter, as described above, is a special case of the general dependency-directed backtracking procedure mentioned earlier. This interpreter makes it possible to gain the efficiency of dependency-directed backtracking automatically while writing search programs in SCHEMER. A more detailed description of the above process can be found in [Zabih *et al.* 1987].

V. Comparison with Other Work

SCHEMER is interesting because it provides automatic backtracking, without specifying a backtracking strategy, in a language that is almost Scheme. It can thus give the user dependency-directed backtracking in a highly transparent manner. Previously available methods for obtaining dependency-directed backtracking include the direct use of a *Truth Maintenance System* (or TMS) [Doyle 1979], deKleer's consumer architecture [deKleer 1986] and the language AMORD [deKleer *et al.* 1978]. These methods, however, require the user to explicitly use dependency-directed backtracking or to write in an unconventional language. They also necessitate special programming techniques, because of the way they use the underlying TMS.

In particular, these methods force the user to provide node labeling and dependency analysis. Deciding which facts in the search problem should be assigned TMS nodes corresponds to node labeling. Providing the TMS with logical implications, so that it can determine the labels responsible for failures, corresponds to dependency analysis. If these implications are not carefully designed it is possible to overlook the contributions of some labels; this can result in unsound nogoods which prune solutions, as mentioned earlier.

Using a TMS directly does not provide a separate language layer at all. It is easy for the problem solver to neglect to inform the TMS of the labels responsible for some decision, leading to unsound nogoods. This is also inconvenient; the user must intersperse code to solve the search problem with calls to the TMS to ensure dependency-directed backtracking. SCHEMER, on the other hand, enforces a clean separation between the code that defines the search problem, which the user writes in SCHEMER, and the code that implements the search strategy, which the interpreter provides transparently.

AMORD provides a language layer, as does the consumer architecture (to a lesser extent). The language is rule-based, though, and thus lacks a single locus of control. Such an approach is well-suited to problems that can be easily expressed with rules and a global database of assertions. On the other hand, it is difficult to use on problems that are not easily converted into rule-based form. A major advantage of SCHEMER is that it allows the user to express search problems without forcing him to think in terms of a rule-set and a global database.

Prolog [Warren *et al.* 1977] is defined to provide depth-first chronological backtracking. However, there has been a fair amount of work on non-chronological backtracking strategies within the Prolog community [Bruynooghe and Pereira 1984]. While it is likely that much of our framework for providing dependency-directed backtracking could be applied to Prolog, we have not yet done so. Complicating matters are several differences between SCHEMER and the "functional" subset of Prolog (i.e. pure horn clause logic). For example, SCHEMER has closures while Prolog, which uses unification to implement parameter passing, potentially has data flowing both into and out of each parameter.

The closest language to SCHEMER is Dependency-Directed Lisp (DDL), a Lisp-based language invented by Chapman to implement TWEAK [Chapman 1985]. This is not surprising, since SCHEMER is based on DDL. There are two differences between DDL and SCHEMER that are worth describing.

First, DDL used a weaker dependency-directed backtracking strategy than SCHEMER does. DDL would never use a nogood more than once. This was because DDL labels never appeared more than once in the search tree. As a result DDL considers parts of the tree containing only failures, which SCHEMER would prune. This in turn was due to the difficulty of devising a choice-naming scheme that produces repeated labels without destroying the semantics of the language.

In addition, DDL had side-effects. Side-effects complicate dependency analysis by introducing too many dependencies. In SCHEMER, justifications can be computed 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 never part of the justification for 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)`, for example, 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. Our (not very determined) attempts to design such a method for dependency analysis have produced such large nogoods that pruning never occurs.

VI. Conclusions

We have shown that SCHEMER, a non-deterministic language based on Lisp, can elegantly express search problems, and that it can provide automatic dependency-directed backtracking. The resulting interpreter allows users to gain the benefits of this backtracking strategy while writing in a remarkably conventional language. We suspect that many search programs could benefit from dependency-directed backtracking if it were only more accessible. It is our hope that SCHEMER will make dependency-directed backtracking a more popular search strategy in the AI community.

Acknowledgments

Alan Bawden, Mark Shirley and Gerald Sussman helped us considerably with SCHEMER. Phil Agre, Jonathan Rees, Jeff Siskind, Daniel Weise, Dan Weld and Brian Williams also contributed useful insights. John Lamping and Joe Weening read and commented on drafts of this paper.

References

[Abelson and Sussman 1985] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[Bruynooghe and Pereira 1984] Maurice Bruynooghe and Luis Pereira. "Deduction Revision by Intelligent Backtracking". In *Implementations of Prolog*, J. Campbell (editor). Ellis Horwood, Chichester, 1984.

[Chapman 1985] David Chapman. "Planning for Conjunctive Goals". MIT AI Technical Report 802, November 1985. Revised version to appear in *Artificial Intelligence.*

[Clinger 1982] William Clinger. "Nondeterministic Call by Need is Neither Lazy Nor by Name." *Proceedings of the ACM Conference on LISP and Functional Programming*, 226–234, 1982.

[deKleer 1986] Johan deKleer. "Problem Solving with the ATMS". *Artificial Intelligence* 28(1986), 197–224.

[deKleer *et al.* 1978] Johan deKleer, Jon Doyle, Charles Rich, Guy Steele, and Gerald Jay Sussman. "AMORD, a Deductive Procedure System". MIT AI Memo 435, January 1978.

[Doyle 1979] Jon Doyle. " A Truth Maintenance System". *Artificial Intelligence* 12(1979), 231–272.

[McCarthy 1963] John McCarthy. "A basis for a mathematical theory of computation". In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (editors). North-Holland, Amsterdam, 1963.

[Rees *et al.* 1986] Jonathan Rees et. al. "Revised^3 Report on the Algorithmic Language Scheme". SIGPLAN Notices 21(12), December 1986.

[Stallman and Sussman 1977] Richard Stallman and Gerald Jay Sussman. "Forward Reasoning and Dependency Directed Backtracking in a System for Computer-Aided Circuit Analysis". *Artificial Intelligence* 9(1977), 135–196.

[Warren *et al.* 1977] D. Warren, L. Pereira and F. Pereira. "Prolog — the language and its implementation compared with Lisp". *ACM Symposium on Artificial Intelligence and Programming Languages*, 1977.

[Zabih 1987] Ramin Zabih. *Dependency-Directed Backtracking in Non-Deterministic Scheme.* M.S. thesis, MIT Department of Electrical Engineering and Computer Science, January 1987. Revised version available as MIT AI Technical Report 956, July 1987.

[Zabih *et al.* 1987] Ramin Zabih, David McAllester and David Chapman. "Dependency-Directed Backtracking in Non-Deterministic Scheme". To appear in *Artificial Intelligence.* (Preliminary draft available from the authors).