Screamer: A Portable Efficient Implementation of Nondeterministic Common Lisp

Jeffrey Mark Siskind*
University of Pennsylvania
Institute for Research in Cognitive Science
3401 Walnut Street Room 407C
Philadelphia PA 19104
215/898–0367
internet: Qobi@CIS.UPenn.EDU

David Allen McAllester†
M. I. T. Artificial Intelligence Laboratory
545 Technology Square Room NE43–412
Cambridge MA 02139
617/253–6599
internet: dam@AI.MIT.EDU

July 11, 1993

Abstract

Nondeterministic LISP is a simple extension of LISP which provides automatic backtracking. Nondeterminism allows concise description of many search tasks which form the basis of much AI research. This paper discusses SCREAMER, an efficient implementation of nondeterministic LISP as a fully portable extension of COMMON LISP. In this paper we present the basic nondeterministic LISP constructs, motivate the utility of the language via numerous short examples, and discuss the compilation techniques.

*Supported in part by an AT&T Bell Laboratories Ph.D. scholarship to the author, by a Presidential Young Investigator Award to Professor Robert C. Berwick under National Science Foundation Grant DCR–85552543, by a grant from the Siemens Corporation, and by the Kapor Family Foundation. Also supported in part by ARO grant DAAL 03–89–C–0031, by DARPA grant N00014–90–J–1863, by NSF grant IRI 90–16592, and by Ben Franklin grant 91S.3078C–1

†Supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-91-J-4038.

1 Introduction

Nondeterminism is a useful programming paradigm popularized by PROLOG. It has long been established in programming language folklore that nondeterminism simplifies the task of writing combinatorial search programs. While straightforward blind search techniques normally associated with nondeterministic programming may not yield acceptable performance, nonetheless the simple task of specifying a combinatorial search space, for smaller more tractable problems, is almost always easier when one avails oneself of nondeterministic constructs than when one uses more conventional programming techniques.

Nondeterminism however, need not remain the sole province of logic programming languages. Any programming language can be extended to support nondeterministic search with the addition of two new constructs: a choice point operator and a failure operator. We have done precisely this for LISP. This paper describes SCREAMER, our implementation of nondeterministic COMMON LISP. Nondeterministic LISP is not new (cf. McCarthy 1963, Clinger 1982, Chapman 1985, Zabih 1987, Zabih et al. 1987, Haynes 1987). What is new with our work however, is a portable and efficient implementation. Most prior implementations were complete custom implementations of a nondeterministic evaluator. In contrast, SCREAMER is implemented as a fully portable macro package which can run under any implementation of COMMON LISP. This allows SCREAMER programs to interoperate in the same environment as other COMMON LISP programs and to leverage off of the rich set of programming language features and the comprehensive programming environments available with COMMON LISP. Furthermore—for reasons to be discussed later in the paper—prior implementations were inefficient. This inefficiency precluded using these implementations for practical programming. SCREAMER on the other hand, has been in regular use by numerous people for several years now, both as a basis for teaching AI and as a substrate supporting current AI research. This paper focuses on three topics. First, it proposes nondeterminism as a useful and *expressive* construct which encourages a simpler and clearer programming style in LISP. We will attempt to illustrate this via numerous examples. Second, it contrasts nondeterministic LISP with PROLOG, illustrating some ways that the former is more expressive than the latter. Finally, it discusses the compilation techniques which allow SCREAMER to generate efficient code.

2 Nondeterministic Lisp

At its core, SCREAMER adds only two new constructs to COMMON LISP. The macro `either` nondeterministically evaluates one of an arbitrary number of subexpressions and returns the result. Operationally, this can be viewed as introducing a choice-point. The expression `(either` e_1 $e_2 \ldots e_n$`)` first evaluates e_1 and returns its result. If the computation fails—either during the evaluation of e_1 or during subsequent computation using the value produced by e_1—the computation backtracks to evaluate e_2 and return its value instead. Failures are introduced by calling the function `fail`, the second primitive construct provided by SCREAMER. Repeated failures cause the evaluation of subsequent subexpressions in the dynamically nested `either` expression until no further subexpressions remain. Choice-points are dynamically nested. Failing to a choice-point whose alternatives are exhausted will propagate to the next most recent choice-point.

The utility of these two constructs is illustrated by the following example. The following function nondeterministically returns an integer between the given bounds. (This function is so ubiquitous that it is built into SCREAMER.)

```
(defun an-integer-between (low high)
 (if (> low high) (fail))
 (either low (an-integer-between (1+ low) high)))
```

Given the nondeterministic function `an-integer-between`, one can write the following procedure to find Pythagorean triples.

```
(defun pythagorean-triples (n)
 (all-values
  (let ((a (an-integer-between 1 n))
        (b (an-integer-between 1 n))
        (c (an-integer-between 1 n)))
   (unless (= (+ (* a a) (* b b)) (* c c)) (fail))
   (list a b c))))
```

This procedure deterministically returns a list of all Pythagorean triples of integers between 1 and n. This example introduces `all-values`, another primitive construct provided by SCREAMER. `All-values` repeatedly evaluates the nondeterministic expression in its body gathering all of its values into a list which is returned deterministically. It is thus analogous to the `bag-of` primitive in PROLOG. SCREAMER also provides a similar primitive, `one-value`, which deterministically returns only the first value computed by its nondeterministic body.[1] `One-value` is similar in many ways to the cut primitive in PROLOG.

The previous example illustrates a typical nondeterministic programming cliché, namely *generate-and-test*. The calls to `an-integer-between` constitute the generator[2] while the expression

```
(unless ... (fail))
```

constitutes the test. While it is easy to formulate generate-and-test procedures in nondeterministic LISP, such procedures are often inefficient. Nondeterministic constructs like `either` and `fail`, however, are more general, and support many other more efficient programming clichés. Consider for example, the N-Queens problem. A generate-and-test solution would first generate arrangements containing N queens and then filter out those arrangements where some queen was under attack. A more efficient solution would interleave the generate and test phases, testing each queen for attacks as it was placed. The code in figure 1 illustrates how this can be formulated in nondeterministic LISP.

[1] SCREAMER adopts a depth-first left-to-right traversal of the search tree when enumerating values of nondeterministic expressions. Thus `all-values` and `one-value` have well-defined denotations. Furthermore, users can (and often must) rely on divergence properties of the search order when writing SCREAMER programs which specify infinite search trees.

[2] We adopt the (unenforced) convention that the names of all generator functions begin with the prefix `a-` or `an-`.

```
(defun attacks? (qi qj distance)
 (or (= qi qj) (= (abs (- qi qj)) distance)))

(defun check-queens (queen queens &optional (distance 1))
 (unless (null queens)
  (if (attacks? queen (first queens) distance) (fail))
  (check-queens queen (rest queens) (1+ distance))))

(defun n-queens (n &optional queens)
 (if (= (length queens) n)
     queens
     (let ((queen (an-integer-between 1 n)))
      (check-queens queen queens)
      (n-queens n (cons queen queens)))))
```

Figure 1: A Screamer program for solving the N-Queens problem.

3 Combinatorial Programming

Many programming tasks involve enumerating the elements of a combinatorial structure such as the subsets or partitions of a given set. While enumerating large combinatorial structures may be intractable, many practical programming tasks require enumerating small combinatorial structures. For such tasks, the prime problem is one of programming convenience, not efficiency. Writing correct combinatorial programs can be an arduous, error prone task. We claim that nondeterministic Lisp allows more transparent specification of combinatorial programs yielding programs which are easier to write, understand, and debug than equivalent deterministic functional programs. We offer the following examples in support of this claim.

Consider the task of enumerating the power set of a given set. This can be accomplished by the following function which nondeterministically returns a subset of a given set.

```
(defun a-subset-of (x)
 (if (null x)
     nil
     (let ((y (a-subset-of (rest x)))) (either (cons (first x) y) y))))
```

Given the above function, the power set of x can be computed by evaluating

```
(all-values (a-subset-of x)).
```

Or consider the task of enumerating the set of all partitions of a given set. To solve this task we first must define the following function which nondeterministically returns a member of a given set. (Again, this function is so ubiquitous that it is built into Screamer.)

```
(defun a-member-of (x)
 (if (null x) (fail))
 (either (first x) (a-member-of (rest x))))
```

As an aside, `a-member-of` and `all-values` are duals of each other. `A-member-of` converts a spatial representation of a set of choices into a temporal one based on backtracking, while `all-values` converts the temporal backtracking representation of a set of choices into a list represented spatially. Given the function `a-member-of`, the following function nondeterministically returns a partition of a given set.

```
(defun a-partition-of (x)
 (if (null x)
     nil
     (let ((y (a-partition-of (rest x))))
      (either (cons (list (first x)) y)
              (let ((z (a-member-of y)))
               (cons (cons (first x) z)
                     (remove z y :test #'eq :count 1)))))))
```

This function operates by taking the elements of x, one at a time, and nondeterministically either placing them in a new partition or in one of the existing partitions.

4 Local Side Effects

All of the examples presented so far could have been written in PROLOG. While we believe that nondeterministic functional programs demonstrate their intent more clearly than their logic program counterparts, the primitives added by SCREAMER, namely `either`, `fail`, `all-values`, `one-value`, and `for-effects` all have analogs in PROLOG. In this section we discuss one further primitive construct added by SCREAMER which has no analog in PROLOG: local side effects.

Adding nondeterminism to LISP raises an important design decision: should side effects be undone upon backtracking? It turns out that it is useful to have two types of side effects, those that are undone upon backtracking and those that are not. SCREAMER supports both types under user control. The former are termed *local* side effects while the latter are termed *global*. Global side effects are useful for gathering statistics about a search process, or for passing information between different branches of a search tree that may aid in pruning future branches. This section will demonstrate the expressive programming power afforded by local side effects.

Consider the problem of enumerating all simple paths between two vertices in a directed graph. A simple path is one which does not visit any vertex more than once. Solving this task will require keeping track of which vertices have been visited in the path currently being constructed. This can be accomplished most efficiently by associating a `visited?` flag with each vertex. Paths are enumerated by starting with the empty path emanating from the source vertex and continually augmenting this path with a neighbor of the vertex currently at the head of the path, until the path contains the sink vertex. Choosing which vertex to add to the path may be nondeterministic since a given vertex may have more than one neighbor. As a vertex is added to the current path, its `visited?` flag is set. A path cannot be augmented with a vertex whose `visited?` flag is already set. Setting the `visited?` flag must be performed by local side effect to allow proper enumeration of alternate paths by backtracking. The above algorithm is captured by the following function which nondeterministically returns a simple path from vertex u to v.

```
(defstruct (node (:conc-name nil)) next-nodes (visited? nil))

(defun simple-path (u v)
 (if (visited? u) (fail))
 (local (setf (visited? u) t))
 (either (progn (unless (eq u v) (fail)) (list u))
         (cons u (simple-path (a-member-of (next-nodes u)) v))))
```

The efficiency of this algorithm depends crucially on the ability to check and update the `visited?` status of a vertex in constant time. Without local side effects this could be done in linear time, by passing around a list of visited vertices and continually checking for membership in that list, or in logarithmic time, by using balanced binary trees to represent the visited vertex list, but not in constant time. The important point here is that the solution based on local side effects is both more efficient and more transparent than either solution not using side effects. PROLOG—lacking the capability for local side effects—could express only the inelegant solutions not using side effects.

One may raise an objection to the above claim that PROLOG lacks the capability for local side effects. Unification of logic variables provides a form of local side effect. Using extra-logical extensions to PROLOG, one could implement a `visited?` flag as an unbound logic variable. Binding that variable would set the `visited?` flag, while checking whether it was bound could be accomplished via the extra-logical `var` primitive. This solution—while of dubious clarity—still does not afford the full generality of local side effects in SCREAMER. PROLOG logic variables are single assignment while SCREAMER allows repeated local side effects to the same variable. The following example illustrates the utility of multiple assignment local side effects.

Let us define a k-simple path as one which does not contain any vertex more than k times. Consider the task of enumerating all k-simple paths between two vertices in a graph for some fixed k. This can be accomplished by maintaining a `visits` count for each vertex instead of a `visited?` flag, incrementing that count each time the vertex is added to the path, and checking that the count is less than k before adding it to the path. This algorithm is illustrated by the following code fragment.

```
(defstruct (node (:conc-name nil)) next-nodes (visits 0))

(defun k-simple-path (u v k)
 (if (= (visits u) k) (fail))
 (local (incf (visits u)))
 (either (progn (unless (eq u v) (fail)) (list u))
         (cons u (k-simple-path (a-member-of (next-nodes u)) v k))))
```

Note that this algorithm will require multiple assignment local side effects to the `visits` count and thus could not be accomplished as efficiently or transparently in PROLOG.

Also note that—as the above examples demonstrate—SCREAMER supports local side effects not only to variables but also to slots of data structures created by `defstruct`. The SCREAMER local side effect mechanism is productive in that it applies to all side effects which can be introduced with `setf` and `setq` including, for example, side effects performed on array elements, hash tables, and CLOS instance slots. Furthermore, a SCREAMER `local` declaration declares all side effect expressions nested lexically in its body to be local, including those introduced implicitly via macros. This allows one to use existing COMMON LISP iteration macros—such as `dolist`—within nondeterministic expressions, simply by wrapping a call to the iteration macro with `local` to convert the side effects to the iteration variable into local ones. Thus the following expression takes a list of lists l, and nondeterministically returns a list containing one element from each list.

```
(let ((a nil))
 (local (dolist (x l) (push (a-member-of x) a)))
 (reverse a))
```

This all points to a methodological bias of our work. PROLOG and COMMON LISP provide orthogonal sets of features. On one hand, PROLOG provides nondeterminism, unification, logic variables, and pattern directed invocation, useful features lacking in COMMON LISP. On the other hand, COMMON LISP provides numerous useful features lacking in PROLOG, most notably data structures and iteration. Rather than arguing for the merits of one language over the other, a language merging the features of both PROLOG and COMMON LISP would be better than either in isolation. Such a language can be arrived at either by adding the missing features of COMMON LISP into PROLOG or vice versa. We have adopted the latter tactic in our work. A key claim whose validity is demonstrated by our work is that it is possible to add nondeterminism to COMMON LISP in a way which is fully portable across all COMMON LISP implementations, does not require any modification to the underlying implementation, and does not suffer performance penalties. The next section illustrates how this is accomplished. A companion paper (Siskind and McAllester 1993) demonstrates how to add unification and logic variables—as well as a complete constraint logic programming package—on top of this basic facility supporting nondeterminism.

5 Implementation

In languages with nondeterminism and automatic backtracking, the occurrence of a failure may require restarting the computation at a point which is no longer on the traditional control stack. Thus, backtracking requires the maintenance of failure continuations. This would be straightforward in a language with first class continuations, such as SCHEME or SML. Since COMMON LISP does not have first class continuations, we construct continuations by performing CPS conversion on SCREAMER programs. For example, SCREAMER converts the following definition for `a-member-of`

```
(defun a-member-of (x)
 (if (null x) (fail))
 (either (first x) (a-member-of (rest x))))
```

into the following CPS converted form.

```
(defvar *fail* #'(lambda () (error "Top level fail")))

(defun fail () (funcall *fail*))

(defun a-member-of′ (c x)
 (if (null x) (fail))
 (let ((*fail* #'(lambda () (a-member-of′ c (rest x)))))
  (funcall c (first x))))
```

Note that the continuation is the first argument to allow nondeterministic functions full use of COMMON LISP argument passing capabilities. Naive CPS conversion introduces a large number of lexical closures not present in the original code. This can result in a severe performance penalty. Accordingly, SCREAMER takes great pains to use CPS conversion sparingly. The SCREAMER implementation does a global static analysis to find code fragments which are provably deterministic and avoids CPS converting those fragments. In general it is not possible to statically determine whether or not a given code fragment is deterministic. Therefore, SCREAMER conservatively finds only a subset of the deterministic code fragments. One can apply varying degrees of sophistication in identifying deterministic code fragments. Considerable effort has been invested to provide SCREAMER with powerful but efficient static analysis.

In spite of the fact that SCREAMER's compilation techniques require global analysis, SCREAMER does support incremental redefinition of procedures. SCREAMER maintains a who-calls database to identify those code blocks requiring recompilation. Thus if f, g, and h are initially deterministic—and f calls g which in turn calls h—redefining h to be nondeterministic will cause SCREAMER to automatically recompile f and g as well, after performing the appropriate CPS conversion. The full paper will explain both the static analysis and incremental recompilation in greater detail.

6 Related Work

Nondeterministic LISP is not new. The addition of a nondeterministic choice operator (sometimes called `amb` or `choose`) to LISP dates back to McCarthy (1963). Clinger (1982) discusses the difficulties involved in giving a formal semantics to a nondeterministic choice operator in LISP. (Chapman unpublished) describes DEPENDENCY DIRECTED LISP (also known as DDL), an implementation of nondeterministic LISP used to implement TWEAK (Chapman 1985), a non-linear constraint-posting planner. DDL recorded dependency information during execution to support selective backtracking. Zabih (1987) and Zabih et al. (1987) describe SCHEMER, an interpreter for nondeterministic SCHEME that recorded and analyzed dependency information to perform both selective backtracking and lateral pruning. SCHEMER and DDL were both interpreters to support retaining the dependency information needed for intelligent backtracking. Haynes (1987) describes how a nondeterministic choice operator can be added to SCHEME using the `call/cc` function.

The techniques used for implementing backtracking in SCREAMER are analogous to those used when compiling PROLOG into LISP (Kahn 1982, 1983, Kahn and Carrlson 1984, Siskind 1989). CPS conversion was used in the RABBIT compiler for SCHEME (Steele and Sussman 1976, Steele 1976, 1978).

7 Summary

Prior implementations of nondeterministic Lisp were too inefficient to be used for practical applications. Screamer remedies this inefficiency by making several design decisions differently than prior implementations. First, Screamer operates as a Common Lisp source to source transform allowing the resulting Common Lisp code generated by the CPS converter to be compiled by the underlying Common Lisp implementation. In contrast, most prior implementations were interpreters. Second, Screamer foregoes intelligent backtracking in favor of chronological backtracking. Experience has shown that the overhead of maintaining dependencies to support intelligent backtracking costs far more than the computation saved in most practical applications. We believe that chronological backtracking—represented via nondeterminism—is a useful programming paradigm when used properly. It is an important control construct which adds significant expressive power to Common Lisp, making programs easier to write, debug, and understand.

We have demonstrated the power of Screamer by using it as a vehicle for teaching 6.824 and CIS520, the graduate core AI courses at M. I. T. and the University of Pennsylvania. As problem sets in these courses, students have used Screamer to build small working versions of a number of programs which have been the focus of AI research in the past and present. These problem sets and other AI course material based on Screamer includes crossword puzzle solvers, Waltz line labeling, Allen's temporal logic, hardware fault diagnosis, A^* search, linear and non-linear planners, natural language query processors based on Montague grammar, Prolog interpreters and compilers, theorem provers based on semantic tableaux, congruence closure, resolution, and Prolog technology, Rete, qualitative simulation, robot path planning, and model-based vision. All of these examples can be written much more clearly and concisely using nondeterministic constructs than without. Furthermore, researchers at M. I. T., the University of Pennsylvania, and several other institutions world-wide have begun using Screamer as part of their ongoing research programming activity.

The current version of Screamer is available by anonymous FTP from the file `ftp.ai.mit.edu:/com/ftp/pub/screamer.tar.Z`. We encourage you to obtain a copy of Screamer and give us feedback on your experiences using it.

References

[1] David Chapman. Dependency-Directed Lisp. Unpublished manuscript received directly from author.

[2] David Chapman. Planning for conjunctive goals. Master's thesis, Massachusetts Institute of Technology, January 1985. Also available as M. I. T. Artificial Inteligence Laboratory Technical Report 802.

[3] W. Clinger. Nondeterministic call by need is neither lazy nor by name. In *Proceedings of the ACM Conference on* Lisp *and Functional Programming*, pages 226–234, 1982.

[4] Christopher T. Haynes. Logic continuations. *Journal of Logic Programming*, 4:157–176, 1987.

[5] Kenneth M. Kahn. A partial evaluator of Lisp written in Prolog. In *Proceedings of the First Logic Programming Conference*, Marseille, France, 1982.

[6] Kenneth M. Kahn. Unique features of Lisp machine Prolog. UPMAIL Report 14, University of Uppsala, Sweden, 1983.

[7] Kenneth M. Kahn and M. Carlsson. How to implement Prolog on a Lisp machine. In J. A. Campbell, editor, *Implementations of* Prolog, chapter 2, pages 117–134. Ellis Horwood, Chichester, 1984.

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

[9] Jeffrey Mark Siskind. The culprit pointer method for selective backtracking. Master's thesis, Massachusetts Institute of Technology, January 1989.

[10] Jeffrey Mark Siskind and David Allen McAllester. Nondeterministic Lisp as a substrate for constraint logic programming. In *Proceedings of the Eleventh National Conference on Artifical Intelligence*, July 1993.

[11] Guy Lewis Steele Jr. Lambda, the ultimate declarative. A. I. Memo 379, M. I. T. Artificial Intelligence Laboratory, November 1976.

[12] Guy Lewis Steele Jr. Rabbit: A compiler for Scheme. Technical Report 474, M. I. T. Artificial Intelligence Laboratory, 1978.

[13] Guy Lewis Steele Jr. and Gerald Jay Sussman. Lambda, the ultimate imperative. A. I. Memo 353, M. I. T. Artificial Intelligence Laboratory, March 1976.

[14] Ramin D. Zabih. Dependency-directed backtracking in non-deterministic Scheme. Master's thesis, Massachusetts Institute of Technology, January 1987.

[15] Ramin D. Zabih, David Allen McAllester, and David Chapman. Non-deterministic Lisp with dependency-directed backtracking. In *Proceedings of AAAI–87*, pages 59–64, July 1987.