A Constructor For Applicative Multiprogramming*

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT NO. 80

A CONSTRUCTOR FOR

APPLICATIVE MULTIPROGRAMMING

DANIEL P. FRIEDMAN

DAVID S. WISE

(REVISED) NOVEMBER, 1979

*Research reported herein was supported (in part) by the National Science Foundation under grants numbered MCS75-06678 A01 and MCS77-22325.

A Constructor for Applicative Multiprogramming*

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana

Abstract

A new structure, the nondeterministically ordered multiset, and a new structure builder for it, frons, are defined as convenient primitives for applicative languages like pure LISP. Since the order of a multiset will be determined as the structure is turned into a list (rather than as the multiset is constructed), functionality or reproducibility of results must be sacrificed in a limited way. This paper is oriented toward the motivation for imbedding nondeterminism in data structures--as opposed to control structure or parameter-passing mechanisms. The heart of the airline reservation system is included.

*Research reported herein was supported (in part) by the national Science Foundation under grants numbered MCS75-06678 A01 and MCS77-22325.

Introduction

Nondeterministic programming is not easily treated in any higher level language. Yet it plays an important role in the design of operating systems, real-time control, and asynchronously parallel systems. It has attracted much attention from software engineers, much programming talent from software shops, and much aggravation from irreproducible events. With processors and parallelism becoming cheaper and ubiquitous, the control of this sort of programming process has become very important. Various attempts to refine the control of interprocess communication [5,8,9,20,21,34] and to extend program proving techniques [10,17,26,27,32] indicate the importance and the limits of this activity under the von Neumann [2] style languages. It has become more apparent that current solutions are not working out and that the impasse is not the fault of the problem. An applicative approach to all programming problems has recently been advanced by Backus [2] in his Turing lecture. The interested reader is referred to his thorough review. His arguments, running orthogonal to current machine architecture, emphasize a purely functional style of programming in which the only binding is of positions within structures to values and the only control structure is function application.

Using only lambda-bindings, function application, conditional expressions, and trivial pimitives, we developed a new perspective on the constructor function which had been used to build all data structures [11]. This paper introduces a new constructor, frons, which allows for the postponement of both the content (as in cons) and the order of a structure. Here the same accessing functions are used to probe into the unordered structure as into an ordered sequence; in fact the user cannot distinguish between such types once it is built. The probing of an unordered structure not only causes the content to appear (as with cons), but it also begins to produce some order within the structure; probing makes a structure behave as if it were ordered.

This nondeterministic structure is called a "multiset" [24]. From the coarse characterization of the multiset above, it is clear that the eventual order might not meet the requirements of stringent functionality; that is, an identically constructed structure may assume a different order. We ask the indulgence of our more theoretically-minded readers during our separation of applicative data structure construction from the rigors of functional programming. (The distinction has more to do with the standard perspective of data structures than with our view of functionality.) The apparent absence of functionality is quite benign and a fair implementation of the constructor exists [16].

The remainder of this paper is in five parts. The first emphasizes syntax, but offers a rough axiomitization of known primitive operations in our notation. The second introduces the multiset as a broad concept and motivates its application with a simple example. It is followed by a detailed definition of the constructor, frons, which is used to construct multisets. The fourth section presents two examples of its application; the more important one is a real-time interrupt handler which is the heart of a solution for the airline reservation system. The final section reviews the relation of this work with that of others in applicative languages. An appendix is attached which offers an implementation of frons in terms of our syntax and in that of LISP.

Syntax

Let ⊥ (read "bottom") denote an ill-defined or computationally divergent value. The notation x↓ is used to indicate that evaluation of x converges in the recursive function theoretical sense; others would write x ≠ ⊥ for x↓. Later we shall require a strict [11 ,36] two argument "identity" function in order to recover some control over the order of evaluation:

strictify:<x y> ≡ y if x↓ .

We present here some syntactic conventions. A sequence s of elements e_1, e_2, ..., e_k is denoted by $<e_1\ e_2\ \ldots\ e_k>$ $(k \geq 0)$. A sequence evaluates to a list of the values of its respective elements; infinite sequences [13,23] are also possible. An application is denoted by an infix colon which is taken to be a right-associative operator. A function (or functional expression) occurs to the left and its argument appears at its right. We introduce this notation and primitives familiar from pure LISP [29] in the prototype examples below; take s as the sequence given above with k > 0.

first:s = e_1; first:<> = ⊥;

rest:s = $<e_2\ \ldots\ e_k>$; rest:<> = ⊥;

cons:<e s> = $<e\ e_1\ e_2\ \ldots\ e_k>$; cons:<e <>> = <e>;

null:s = false; null:<> = true;

atom:s = false; atom:null:s = true;

eq:<atom:null:s null:s> = false; successor:<6> = 7;

if:<null:s banana first:s> = if null:s then banana else first:s = e_1.

The last line indicates that arguments are called-by-need [37] or called-by-delayed-value [36] (or that we are using a lazy evaluator [18,14]) so that unneeded/undefined ones like *banana* need not be a bother (*banana* is an unbound variable); it also indicates a syntactic convenience for conditionals which we adopt in the examples below. We include the starred form of functional combination [12] which will be extended for multisets later. This is a purely syntactic convention which effects LISP's MAP functions or Backus's α (apply to all) operator [2] when a structure appears in the functional position.

<successor*>:<<1 2 3>> = <2 3 4>;

<f*>:<<a b c><x y z>> = <f:<a x> f:<b y> f:<c z>>;

<f*>:<s rest:s> = <f:<e_1 e_2> f:<e_2 e_3> ... f:<e_{k-1} e_k>>.

A starred function is spread across its arguments, and because we allow infinite structures, an infinite spreading is possible:

naturals = cons:<1 <successor*>:<naturals>> = <1 2 3 4 ... .

Rather than use explicit "lambda" conventions we shall define new functions from old ones by a pattern equivalent to a prototype invocation:

or:disjuncts ≡
  if null:disjuncts *then* *false*
  else *if* first:disjuncts *then* *true*
  else or:rest:disjuncts .

Thus, or:<atom:s null:s> = *false*;

or:<atom:s null:s atom:null:s> = *true*;

and or:<atom:s first:<> null:<>> = $\perp$.

Multisets

The braced sequence $m = \{e_1\ e_2\ \ldots\ e_k\}$ denotes a multiset of elements e_i for $1 \le i \le k$ where it is not necessary that $e_i\downarrow$ for every i. In light of this possibility we want to define first and rest on multisets as a generalization of this behavior on sequences.

first:m = e_j where $1 \le j \le k$ is chosen so that $e_j\downarrow$; first:{} = $\perp$;
rest:m = $\{e_1\ e_2\ \ldots\ e_{j-1}\ e_{j+1}\ \ldots\ e_k\}$ where j is chosen as above.
rest:{} = $\perp$.

Note that we do not consider whether or not $e_i\downarrow$ for $i \ne j$; j is chosen without regard to that.

null:m = false; null:{} = true; atom:m = false.

These last rules demonstrate that "{}" = "<>" within the user's semantics specified by these primitives.
(The reader is referred to a more complete presentation of these semantics [15] for a definition of the behavior of cons and frons (i.e. multisets) in building a structure which is something between a sequence and a multiset.) From these prototypes we can derive that or:{atom:s null:s atom:null:s} = true;

or:{atom:s first:<> null:<>} = true;

and or:{rest:{} first:<> rest:<>} = $\perp$.

These results arise without changing the definition of or†. The old definition [30] is sufficient to define "symmetric" or [31] which converges whenever any disjunct is true or when all are false; we just changed the argument.

---

†The behavior depends much on the binding of the variable disjuncts to a single multiset, whence first and rest must choose the same j. See below.

The angle-bracketed combinator is now simply extended to a braced combinator; the brackets merely determine whether the result is taken as a sequence or as a multiset.

{f*}:<<a b c> <x y z>> = {f:<a x> f:<b y> f:<c z>};

{successor*}:<<1 2 3>> = {2 4 3}.

At this point functionality is set aside. We wish to relax the demands of the semantic sense of functionality ever so carefully so that we can handle nondeterministic problems. This does not mean to imply that we can afford to drop all the trappings of such languages. On the contrary, we want to preserve as many of these properties as possible because the practical benefits of this style of programming may be attributed to the lack of side-effects and the preservation of environments. Therefore, as we introduce nondeterminism we "encapsulate" it so that the desirable properties of determinism in other parts of the language are preserved.

The constructor, frons

Like the sequence constructor cons, we have a multiset constructor frons. In the same way that we understand that

$$s = \text{cons}:\langle e_1\ \text{cons}:\langle e_2\ \ldots\ \text{cons}:\langle e_k\ \langle\rangle\rangle\ \ldots\ \rangle\rangle$$

we use the new constructor to build the multiset m from above

$$\text{frons}:\langle e_1\ \text{frons}:\langle e_2\ \ldots\ \text{frons}:\langle e_k\ \{\}\rangle\ \ldots\ \rangle\rangle$$

but since order doesn't matter we also might build m as

$$\text{frons}:\langle e_k\ \ldots\ \text{frons}:\langle e_2\ \text{frons}:\langle e_1\ \{\}\rangle\rangle\ \ldots\ \rangle$$

or in many other ways. The rules for coarse interaction (that is, in the absence of bindings) between frons and the other primitives follow.

$$\text{first}:\text{frons}:\langle x\ y\rangle = \begin{cases} x \text{ if } x{\downarrow}; & (1) \\ \text{first}:y \text{ if first}:y{\downarrow}. & (2) \end{cases}$$

$$\text{rest}:\text{frons}:\langle x\ y\rangle = \begin{cases} y \text{ if } x{\downarrow}; & (3) \\ \text{frons}:\langle x\ \text{rest}:y\rangle \text{ if first}:y{\downarrow}. & (4) \end{cases}$$

$$\text{null}:\text{frons}:\langle x\ y\rangle = \underline{\text{false}};\quad \text{atom}:\text{frons}:\langle x\ y\rangle = \underline{\text{false}}.$$

Because frons behaves so much like cons, the user has no way of distinguishing how a structure was built after it is built; for access purposes a multiset behaves as if it were a sequence. Indeed, the rules for frons are an extension of the rules for cons [29, 11] since cons is defined by alternatives (1) and (3) only; cons has no choice. In the definitions above there is a potentially dangerous freedom of choice when both x↓ and first:y↓, because the choice of alternatives may be different in the application of first and rest. That is, we might choose $\text{first}:\text{frons}:\langle x\ y\rangle = \text{first}:y$ (2)

and rest:frons:<x y> = y (3) so that x might not be in the multiset frons:<first:frons:<x y> rest:frons:<x y>> ! The picture is far different, however, if the multiset has already been constructed and bound to a variable, because such choices on a bound variable are immutable.

Let the binding of the variable v be frons:<x y> (i.e we fix an occurrence of frons:<x y>). Then we define

$$\text{first:v} = \begin{cases} \text{x if x}{\downarrow}\text{ and the other alternative has not yet been chosen for v;} & (5)\\[1ex] \text{first:y if first:y}{\downarrow}\text{ and the other alternative has not yet been chosen for v.} & (6) \end{cases}$$

$$\text{rest:v} = \begin{cases} \text{y if x}{\downarrow}\text{ and the other alternative has not yet been chosen for v;} & (7)\\[1ex] \text{fons:<x rest:y> if first:y}{\downarrow}\text{ and the other alternative has not yet been chosen for v.} & (8) \end{cases}$$

In the case that both x↓ and first:y↓, the choice of alternatives is not so free as above because any previous probing of v may have already determined it. This predetermination of first:v ((5) or (6)) also arises if the other alternative ((7) or (8), respectively) had been chosen as a result of a rest:v probe, and vice versa. The net result is that every variable (or field of a structure) bound to a single multiset must yield consistent results under first and rest (i.e. first and rest behave functionally as long as they are invoked over the same bindings). This must extend to all other variables or fields bound to the same multiset in this or other environments. (Bindings arise from argument-parameter assignment upon function application; they may not be based on lexical matching when frons is involved.)

This convention on binding applies to braced structures as well: the choice of j for first:m must be made consistently with the choice for rest:m (in the example which introduced braces) because of the binding of m. Similarly, the lambda-binding of disjuncts causes the symmetric or example to work properly. Operationally, every incarnation of a multiset is obliged to keep track of which alternative has been chosen after the choice is made, because its behavior must be immutable. The difference between these rules (5-8) and those above (1-4) is that in the earlier case the explicit constructor created a new (and unshared) multiset so that there were no previous choices possible. The appendix presents an implementation of frons which guarantees this behavior. Like our definition of cons which allows specification of infinite sequences, that definition allows specification of infinite multisets.

Examples

In order to demonstrate the facility of nondeterministic programming with the new constructor frons, we present a few example programs which solve some problems of nondeterminism in an applicative style. In reading these examples the reader should notice how the nondeterminism is isolated into the data structure, so that the program is rather simple. First we consider the problem of flattening a multiset of sequences, $M = \{s_1\ s_2\ \ldots\ s_k\}$ into a sequence. In this example the argument is much like a matrix except that we allow for infinite bounds (i.e. the number of rows and the number of elements in each row may be infinite), in which case the first two lines of merge are meaningless.

```
merge:M ≡
  if null:M then M
  else if null:first:M then merge:rest:M
  else cons:<first:first:M
             merge:frons:<rest:first:M rest:M>> .
```

The use of the two constructor functions in merge is particularly interesting. Assuming that the argument is defined appropriately, we can interpret the four possible substitutions of the two constructors in those positions. If both were cons, then merge would append [30] all the rows of M in the order they are presented in M; if the first row of M is infinite then that row is copied.

With the constructors as in the definition of merge, the effect is to interleave the various rows of M; so the order of each one is preserved, but elements from other rows may be interspersed in the final result. If both constructors were frons, the effect would be to allow any mixture of all the elements of the array as an ordering in the result. In the unusual case that the first constructor was frons and the second was cons, a similar mixture would result but elements in the result would be restricted to rows only up through the first infinite one in M.

The interleaving behavior is what we desire for the example below. We would like to write a nondeterministic input driver for a time-sharing system. Specifically, we want to solve the input problem for the airline reservation system [7,40]. In that problem we have an arbitrary number of remote agents' terminals each producing an infinite stream of characters. Each stream forms a row of an input matrix. Thus every row is infinite (as time passes); and there are an indeterminate number of rows (new terminals may be activated at any time). The problem is to write an applicative program which will accept these characters as soon as they are typed (regardless of the inactivity of another terminal) and interleave them into a single input stream with each character identified according to its source.

We require an auxiliary function which will transform a file--a sequence of characters-- into a sequence of pairs --a character and the signature of the file. Furthermore, that sequence of pairs, and all its suffixes, should be strict in the convergence of their first characters. That strictness precludes convergence of such sequences until their first character has been typed at the corresponding remote terminal.

```
identify:<file id> ≡
  if null:file then file
  else strictify:<first:file cons:<<first:file id>
                                   identify:<rest:file id>>> .
```

(Even though _identify_ specifies a full computation over _file_, the reader should satisfy himself that each step is suspended until it is needed.)

It is the multiset of identified files which must be merged. Let us assume that _files_ is a sequence of the sequential files to be interleaved. Then we may invoke the function _fanin_ upon _files_ and _naturals_ in order to generate the desired stream of agents' communication:

```
fanin:<files signatures> ≡ merge:{identify*}:<
                                   files
                                   signatures > .
```

The braced combinator is used to convert the sequence of files and signatures into a multiset of "strictified-identified" files. Thus, the application of _first_ in _merge_ can only yield a result from an active teletype.

Relation to other work

There have been many other attempts to introduce nondeterminism in a controlled way into programming languages and models of computation. Of these, we mention only those that are defined for use in languages with "applicative" flavor*. McCarthy [29] suggested a nondeterministic "ambiguity" operator which is clearly not a function.

$$\text{amb}:\langle x\ y\rangle = \begin{cases} x \text{ if } x\downarrow; \\ y \text{ if } y\downarrow. \end{cases}$$

The amb operator implements pure, mindless nondeterminism; results are not necessarily reproducible. Its failing is that there is no way of knowing which argument was chosen and/or recovering the unchosen one in order to resume its computation later. Others have developed variations on amb (e.g. Ward [39] developed a function either [3] which allowed for amb with a finite number of choices) and Hennessey and Ashcroft [19] introduced parameter passing mechanisms for when the choice was to be made: call-time choice and run-time choice. Kosinski [25] introduced an arbiter into the data flow approach [6]. The arbiter merges streams [4,28] nondeterministically resulting in a single new stream. Using the arbiter, Dennis [7] fabricates an airline reservation system. The arbiter is behaviorally similar to the program fanin; but since it is postulated as a hardware primitive, it may only have a finite number of input streams. Arvind and Gostelow [1] have adopted a similar primitive in their data flow model.

---

* Waldinger and Levitt [38] have developed an unordered structure called a bag (see also [33]), but a bag contains only convergent elements.

We have planted all nondeterminism inside data structures. A formalist may not perceive the significance of this choice, but any programmer will; we want to encapsulate nondeterminism where the programmer may plant it and ignore it easily according to his preconceived notions about programming style. <u>A significant contribution of the applicative style to this development is the experience with an applicative regimen which allows the simple introduction of nondeterminism as a trivial twist in programming style</u>. A good programmer follows a few simple rules when he works: don't compute the same thing twice; never build the same structure twice; etc. The use of the second rule is applied when a good programmer borrows a reference to a shared structure rather than copying it. The point here is that the manipulation of data structures has engendered precisely the same programming practice which is required of properly used nondeterminism in an applicative programming language.

<u>Acknowledgement</u>: We are grateful for the contributions of several people to this work, particularly Mitchell Wand for his creative criticisms, Steve Johnson for his insightful implementation [22], and Steve Smoliar for carrying the development of multisets into the practice of software engineering [35].

REFERENCES

1. Arvind and K. Gostelow. Some relationships between asychronous interpreters of a dataflow language. In Formal Description of Programming Concepts, E. Neuhold (ed.), Amsterdam, North-Holland (1978), 92-119.

2. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Comm. ACM 21, 8 (August, 1978), 613-641.

3. H.G. Baker, Jr., and C. Hweitt. The incremental garbage collection of processes. Proc. Symp. on Artificial Intelligence and Programming Languages, ACM SIGPLAN Notices 12, 8 (August, 1977), 55-59.

4. W.H. Burge. Recursive Programming Techniques, Reading, MA, Addison-Wesley (1975).

5. R.P. Case and A. Padegs. Architecture of the IBM Systems/370. Comm. ACM 21, 1 (January, 1978), 73-96.

6. J.B. Dennis. First version of a dataflow language. In Programming Symposium, B. Robinet (ed.), Berlin, Springer (1974).

7. J.B. Dennis. A language design for structured concurrency. In Design and Implementation of Programming Languages, J.H. Williams and D.A. Fisher (eds.), Berlin, Springer (1977), 231-242.

8. J.B. Dennis and E.C. Van Horn. Programming semantics for multiprogrammed computations. Comm. ACM 9, 3 (March, 1966), 143-155.

9. E.W. Dijkstra. Cooperating sequential processes. In Programming Languages, F. Genuys (ed.), New York, Academic Press (1968), 43-112.

10. E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: an exercise in cooperation. Comm. ACM 21, 11 (November, 1978), 966-975.

11. D.P. Friedman and D.S. Wise. CONS should not evaluate its arguments. In Automata Languages and Programming, S. Michaelson and R. Milner (eds.), Edinburgh, Edinburgh Univ. Press (1976), 257-284.

12. D.P. Friedman and D.S. Wise. Functional combination. Computer Languages 3, 1 (1978), 31-35.

13. D.P. Friedman and D.S. Wise. Unbounded computational structures. SOFTWARE--Practice and Experience 8, 4 (July-August, 1978), 407-416.

14. D.P. Friedman and D.S. Wise. A note on conditional expressions. Comm. ACM 21, 11 (November, 1978), 931-933.

15. D.P. Friedman and D.S. Wise. Applicative multiprogramming. Tech. Rept. 72, Computer Science Dept., Indiana University (January, 1978).

16. D.P. Friedman and D.S. Wise. An approach to fair applicative multiprogramming (in preparation).

17. D. Gries. An exercise in proving parallel programs correct. Comm. ACM 20, 12 (December, 1977), 921-930.

18. P. Henderson and J.H. Morris, Jr. A lazy evaluator. Proc. 3rd ACM Symp. on Principles of Programming Languages (1976), 95-103.

19. M.C.B. Hennessy and E.A. Ashcroft. Parameter-passing mechanisms and nondeterminism. Proc. 9th ACM Symp. on Theory of Computing (1977), 306-311.

20. C.E. Hewitt and R. Atkinson. Parallelism and synchronization in actor systems. Proc. 4th ACM Symp. on Principles of Programming Languages (1977), 267-280.

21. C.A.R. Hoare. Communicating sequential processes. Comm. ACM 21, 8 (August, 1978), 666-677.

22. S.D. Johnson. An Interpretative Model for a Language based on Suspended Construction, M.S. thesis, Indiana University (1977).

23. G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In Information Processing 77, B. Gilchrist (ed.), Amsterdam, North-Holland (1977), 993-998.

24. D.E. Knuth. The Art of Computer Programming 2, Seminumerical Algorithms, Reading, MA, Addison-Wesley (1969), 551.

25. P.R. Kosinski. A data flow language for operating systems programming. Proc. ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices 8, 9 (September, 1973), 89-94.

26. L. Lamport. Proving the correctness of multiprocess programs. IEEE Trans. on Software Engineering SE-3, 2 (March, 1977), 125-143.

27. L. Lamport. Time, clocks, and the ordering of events in a distributed system. Comm. ACM 21, 7 (July, 1978), 558-565.

28. P.J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. Comm. ACM 8, 2 (February, 1965), 89-101.

29. J. McCarthy. A basis for a mathematical theory of computation. In Computer Programming and Formal Systems, P. Braffort and D. Hirschberg (eds.), Amsterdam, North-Holland (1963), 33-70.

30. J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.E. Levin. LISP 1.5 Programmer's Manual, Cambridge, MA, M.I.T. Press (1962), Chapter I.

31. Z. Manna. Mathematical Theory of Computation, New York, McGraw-Hill (1974), Chapter 5.

32. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. Acta Informatica 6, 4 (August, 1976), 319-340.

33. L. Robinson and K.N. Levitt. Proof techniques for hierarchically structured programs. Comm. ACM 20, 4 (April, 1977), 271-282.

34. A.C. Shaw. The Logical Design of Operating Systems, Englewood Cliffs, NJ, Prentice-Hall (1974), 80-82.

35. S.W. Smoliar. Using applicative techniques to design distributed systems. Proc. IEEE Conf. on Specifications of Reliable Software (to appear).

36. J. Vuillemin. Correct and optimal implementation of recursion in a simple programming language. J. Comp. Sys. Sci. 9, 3 (June, 1974), 332-354.

37. C. Wadsworth. Semantics and Pragmatics of Lambda-calculus, Ph.D. dissertation, Oxford (1971).

38. R.J. Waldinger and K.N. Levitt. Reasoning about programs. Artificial Intelligence 5, 3 (Fall, 1974), 235-316.

39. S.A. Ward. Functional Domains of Applicative Languages, Ph.D. dissertation, M.I.T. (1974).

40. A. Yonezawa and C.E. Hewitt. Modelling distributed systems. 5th Intl. Joint Conf. on Artificial Intelligence (1977), 370-377.

Appendix

This appendix presents an implementation of the frons constructor assuming only the amb and strictify primitives defined in this paper, and an understanding of call-by-need [37] or call-by-delayed-value [36] parameter linkages. In particular the reader is cautioned that the list constructor cons from LISP and its accessing functions first and rest (i.e. car and cdr) build structures whose contents are only evaluated once, but such an evaluation is postponed until the structure is probed for the first time in a particular field [11], just as the first use of a formal parameter causes evaluation of the corresponding arguments. Johnson [22] has implemented such an interpreter for the language defined here in PASCAL for the CDC CYBER computers.

Thus, the function insulate is a bit more than an identity function on structures.

insulate:<z> ≡ cons:<first:z rest:z> .

Because the argument for the parameter is called-by-need, and because cons does not need it in order to converge, this function always converges before evaluating its argument. Evaluation is postponed until either field of the resulting structure is probed, and then it is only evaluated once.

Then

```
frons:<x y> ≡
  insulate:<amb:<
             strictify:<x cons:<x y>>
             strictify:<first:y cons:<first:y frons:<x rest:y>>> >>.
```

Because *insulate* is called immediately from *frons*, *frons* always converges to yield a *cons* data structure. The content of that structure is not determined until it is probed because *amb* is not invoked until then. Furthermore, the *amb* expression is evalauted at most once for each invocation of *frons*. At the time of that probe (by either *first* or *rest*) one of the choices in *amb* is made and that determines whether Lines (5) and (7) or Lines (6) and (8) will be used to interpret the multiset bound *once* to z.

In addition we offer a *similar* definition for a LISP interpreter which uses call-by-need. An example of such an interpreter appears in the appendix to our earlier paper [11]. The only operations which need be added to McCarthy's pure LISP interpreter [30, Ch. 1] are *amb* [29] and *prog2* [30], which happens to coincide (for the wrong reasons) with *strictify*. (*prog2* is *not* to be implemented using call-by-need although its first argument is never needed; we require strictness.) Then *frons* is

```
(LABEL FRONS (LAMBDA (X Y)
             ((LAMBDA (Z) (CONS (CAR Z) (CDR Z)))
              (AMB (PROG2 X (CONS X Y))
                   (PROG2 (CAR Y) (CONS (CAR Y) (FRONS X (CDR Y))))
              )) )) .
```