A NOTE ON CONDITIONAL EXPRESSIONS*

by

Daniel P. Friedman

And

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 70

A NOTE ON CONDITIONAL EXPRESSIONS

DANIEL P. FRIEDMAN

AND

DAVID S. WISE

OCTOBER, 1977
REVISED: APRIL, 1978

Abstract: Evaluation of a conditional expression may succeed even when the "deciding predicate" diverges and the alternatives are records (or nodes) whose fields have different content.

Keywords and Phrases: parallel evaluation, suspending CONS, LISP, conditional forms, IF-THEN-ELSE, ambiguous function, infinite structures.
CR Categories: 4.2, 4.32, 5.24, 4.13.

---

*Research reported herein was supported (in part) by the National Science Foundation under grants numbered DCR75-06678 and MCS75-08145.

In an early paper [7,p. 55] McCarthy proposed eight equations for the formal properties of conditional forms in pure LISP. Each of these equations are labelled as strong equivalences except the first:

$$\underline{\text{if}}\ p\ \underline{\text{then}}\ a\ \underline{\text{else}}\ a = a\ . \qquad (1)$$

This equation states that when a conditional expression has identical arms then the value is surely *the* value of those arms, independently of the value of the "deciding predicate", p. The weakness of this equation arose because the equation need not hold if evaluation of p does not terminate (diverges). In that case we would like the value of the conditional to be a, if indeed both arms are known to be equal, but the sequential implementation of conditional expressions precludes it (but see [8]).

Since implementations of conditional expressions usually follow the model of conditional statements, they are universally perceived as being sequential. Nearly all implementations require the evaluation of the predicate by the single processor before either arm is evaluated. In the case that the predicate diverges it is sufficient that the conditional expression, itself, diverges.

The resolution of the weakness of (1) lies in its inherent need for suspended (delayed or lazy) evaluation in implementing McCarthy's language [7]. The delayed evaluation work of Vuillemin [10], sidesteps this issue by defining the conditional expression as a primitive operation strict in its first parameter (i.e. the deciding predicate). A function is *strict in its i^{th} parameter* if divergence of the i^{th} argument implies divergence of the function on that list of arguments [2].

Scott [9] even requires that

if ⊥ then a else b

be ⊥ (indicating a divergent computation) in his lattice-theoretic approach to computation. Henderson and Morris [5,p. 103] remark on the need for simultaneous evaluation of all three parts of a conditional expression as an apparent problem for the *really* "lazy evaluator."

In this note we propose an implementation and corresponding equations for a solution to this problem. Two of our four alternatives are identical to two of McCarthy's equations. Our third and fourth alternatives displace McCarthy's weak (1). His remaining four are somewhat altered†, at the cost of some simplification on arithmetic expressions. Difficulties arise when identical divergent computations (in this case the predicates) are represented as two disassociated suspensions [2].

---

† We accept the sufficiency of the remainder but not their necessity. These distributive equations on conditionals may replace his.

```
if (if p then q else r) then a else c ≡
                    if p then (if q then a else c)
                         else (if r then a else c);

if p then (if q then a else b) else c ≡
                    if q then (if p then a else c)
                         else (if p then b else c);

if p then a else (if r then c else d) ≡
                    if r then (if p then a else c)
                         else (if p then a else d).
```

A suspension is a temporary structure planted within the field of a record when it is created instead of the value which rightfully should be there. It contains information sufficient to derive that value at any time it is necessary to the course of the computation. In terms of LISP, sufficient information is the form which specifies the value of the field and the environment which retains all bindings necessary to evaluate that form at any time in the future. Since both these data are readily available at any point within a pure LISP system, creation of a suspension requires only constant time and space. It follows that LISP's record constructor function, cons, is well-defined over any arguments. This redefinition of cons allows for function definitions whose value has undefined substructure or appears to be an infinite list (see also Landin's streams [6]).

For example, we define evens as the infinite list of even integers beginning at 2 and powers as the infinite list of powers of 2 also starting at 2:

```
evens ≡ evennumbers(1) = (2 4 6 8 10 12 ... );
evennumbers(n) ≡ cons( double(n), evennumbers(succ(n)) );
powers ≡ powersof(1) = (2 4 8 16 32 64 ... );
powersof(n) ≡ cons( double(n), powersof(double(n)) ).
```

Accessing the fields of a structure so suspended is not quite as easy as in pure LISP. When the fields of a record are probed with accessing functions (car and cdr in LISP), the presence of a suspension must be detected. If one is present, it must first be coerced. Coercion is evaluation of the retained form in the retained environment and requires time and space no greater than that required if the argument had been evaluated when the record was constructed in the first place. (Probably considerably less,since it stops at the first record construction.) When the value is found it displaces the suspension (as in call-by-delayed-value [10]) to be immediately accessible on future probes on that field, and it is returned as the value of the probe invocation which caused coercion.

We have shown that such an implementation strengthens considerably the classic power of pure LISP. In fact, it is necessary to meet McCarthy's definitions for car, cdr, and cons [7]. We here combine this with another of McCarthy's [7 ,p. 48] operators AMB to strengthen the problematic equation for conditional expressions.

McCarthy defines a binary ambiguity operator AMB(x,y) to yield the value x or the value y when both are defined, to yield the value which is defined if only one is, and to yield undefined results otherwise. An implementation for AMB requires simultaneous evaluation of both arguments to guarantee that the convergent argument is not starved while the divergent one monopolizes the evaluator. The utility of this operator for multiprocessors should not be overlooked. Let us define a four-argument ambiguity operator which we define using braces rather than parentheses to suggest that the argument order is truly irrelevant:

$$\text{AMB4}\{w,x,y,z\} \equiv \text{AMB}(\ \text{AMB}(w,x),\text{AMB}(y,z)\ ).$$

We distill the concept of conditional expression down to the function *guard*. The name of this two parameter function is motivated by Dijkstra [1]. It is strict in its first argument and satisfies two equations similar to McCarthy's second and third equations for conditionals:

$$\begin{array}{l} \text{guard}(\text{True},x) = x; \\ \text{guard}(\text{False},x) = \bot \end{array} \qquad (2)$$

where $\bot$ is not a value which will be chosen by AMB. These equations may be satisfied by implementing

$$\text{guard}(p,x) \equiv \underline{\underline{\text{if}}}\ p\ \underline{\underline{\text{then}}}\ x\ \underline{\underline{\text{else}}}\ \bot$$

under *any* common interpretation of **if-then-else** .

Dijkstra's *guarded-if* may itself be implemented using these mechanisms:

$$\left.\begin{array}{r} \text{guarded-if}(p_1,\ e_1, \\ p_2,\ e_2, \\ p_3,\ e_3, \\ p_4,\ e_4\ ) \end{array}\right\} \equiv \left\{\begin{array}{r} \text{AMB4}\{\text{guard}(p_1,\ e_1), \\ \text{guard}(p_2,\ e_2), \\ \text{guard}(p_3,\ e_3), \\ \text{guard}(p_4,\ e_4)\ \}\ . \end{array}\right.$$

(Guarded conditionals of lengths other than four are clearly available.) The semantics of this definition properly includes Dijkstra's; we allow several p_i to be $\bot$ or even to be *true* with the corresponding $e_i=\bot$, so long as at least one of the p_j is *true* where e_j is defined.

Using AMB4, guard, boolean operators, and known LISP primitives we now define our conditional expression:

```
if p then x else y ≡
    AMB4{guard( p    ,  x ),
         guard( not(p),y ),
         guard( and(atom(x),atom(y),eq(x,y)),x )
         guard( and( not(atom(x)),not(atom(y)) )
                cons( if p then car(x) else car(y),
                      if p then cdr(x) else cdr(y) ))
        } .
```

The four choices for the ambiguity operator correspond to four equations for conditional expressions in pure LISP:

```
if True then x else y = x;
if False then x else y = y;
if p then a else a = a (where a is an atom);            (3)
if p then cons(q,r) else cons(s,t) =
     cons( if p then q else s, if p then r else t )
```

The first two equations above appear in McCarthy's paper[7]. The second two are cases of McCarthy's first equation (1); the third equation reflects his later work with Manna [8].

The suggested implementation of this semantics for

```
E = if p then x else y
```

simultaneously evaluates all of p, x, and y. If p converges first, then the value E will be that of a chosen one of x or y, so one of those evaluations may be abandoned. If both x and y converge first and they are of the same type, then one of the second two equations are applied to yield a value independently of the convergence of p. Significantly no evaluation ever proceeds past the first invocation of cons, and the chance for convergence is thereby considerably enhanced. Furthermore, when the fourth equation is applied, no further computation is implied by the conditional arguments on the right; these forms are immediately subsumed into a suspension.

McCarthy offers a distributive law for function invocation across conditional expressions [7,p 58] which contrasts with our fourth equation. We have instead specified distributing conditional expressions across function invocation, taking advantage of a particularly recognizable invocation. Cons is a distinguished function which leaves its signature on its result so we can easily detect its use post facto using the atom predicate. Regardless of the depth at which cons was invoked in yielding non-atoms for both arms of the conditional, the applicability of the fourth equation is detectable and is valid.

Two examples explain the impact of AMB on our semantics. The first concerns two equal infinite lists, whose equality is not directly computable because of their lengths. Let

```
addtwo*(lis) ≡ cons( succ(succ(car(lis))),
                     addtwo*(cdr(lis)) ).
```

Then the result of

```
if equal(length(evens),length(powers))
   then cdr(evens) else addtwo*(evens)
```

is (4 6 8 10 12 ...) in spite of the fact that the predicate is ⊥. The arms of the conditional are both constructed, and so therefore is the answer. The only equality tested is on the integers in the answer list -- not on the lists themselves.

Consider, secondly, the value of the expression

```
if equal(length(evens),length(powers))
   then evens else powers
```

which is partially defined. It is a list of infinite length whose first two elements are defined. The printed result is

```
(2 4 ???
```

where the ellipsis of question-marks is the message of an evaluator exceeding its allotted resources. In this case the resource is exhausted trying to determine the length of infinite lists, whose convergence would select 6 or 8 as the next element of the result.

The fact that structures being printed are traversed in preorder as they are printed [3] colors this latter example considerably. It requires that we provide sequences with a common prefix, as opposed to a suffix, as the arms of this exemplary conditional expression. The behavior of the evaluator would be the same if the arms were the lists (2 4 6 8 10) and (2 4 91 8 10); the evaluation diverges -- awaiting the value of the predicate -- before the common suffixes are even noticed. If the accessing pattern were random then accesses to all but the third element of the list defined by such a conditional expression would succeed! In contrast, if the arms were respectively (1 2 3) and (2 2 3) then the list printed would have been

```
(???   ;
```

if the structures were a list and an atom, say (2) and 2, then the behavior would have been

```
???
```

without even printing the left parenthesis.

These examples run with the results described on a prototype system, which extends the formalism of unordered structures with divergent elements considerably beyond that suggested by AMB and its derivatives. The formalism and the implementation of the model which led to our note are reported elsewhere [4].

References

1. Dijkstra, E. W. *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ (1976).
2. Friedman, D. P., and Wise, D. S. CONS should not evaluate its arguments. In S. Michaelson and R Milner (eds.), *Automata, Languages and Programming*, Edingurgh Univ. Press, Edinburgh (1976), 257-284.
3. Friedman, D. P., and Wise, D. S. Output driven interpretation of recursive programs, or writing creates and destroys data structures. *Information Processing Lett.* *5*, 6 (Dec., 1976), 155-160.
4. Friedman, D. P., and Wise, D. S. Applicative multiprogramming. Technical Report No. 72, Computer Science Department, Indiana University (1978).
5. Henderson, P., and Morris, J., Jr. A lazy evaluator. *Proc. 3rd ACM Symp. on Principles of Programming Languages* (Jan., 1976), 26-45.
6. Landin, P. J. A correspondence between ALGOL 60 and Church's lambda notation. *Comm. ACM.* *8*, 2 (Aug., 1965), 89-101.
7. McCarthy, J. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg (eds.), *Computer Programming and Formal Systems*, North-Holland, Amsterdam (1963), 33-70.
8. Manna, Z., and McCarthy, J. Properties of programs and partial function logic. In B. Meltzer and D. Michie (eds.), *Machine Intelligence 5*, American Elsevier, New York (1970), 27-37.
9. Scott, D. S. Data types as lattices. *SIAM J. Comput.* *5*, 3 (Sept., 1976), 522-587.
10. Vuillemin, J. Correct and optimal implementation of recursion in a simple programming language. *J. Comp. Sys. Sci.* *9*, 3 (1974), 332-354.