September 1981

CS-TR-81-7

DETECTION OF AVAILABLE CONCURRENCY IN LISP PROGRAMS

by

Jed B. Marti

Department of Comp. and Inf. Science
The University of Oregon
Eugene, Oregon 97403

ABSTRACT A method for monotonic global data flow analysis in LIS programs is presented. It is shown how this analysis can be used t identify available concurrency in the presence of global sid effects, aliasing, and the run time creation of variables.

Key Words and Phrases: data flow analysis, LISP, concurren programming

CR CATEGORIES: 5.24, 4.32, 4.12, 4.13

1.0 INTRODUCTION

In most functional and near functional languages, availabl concurrency can be classified into three categories: *iterative* *compositional* and *horizontal*. Iterative concurrency is generall indicated by a repetitive data structure. Several processors may b applied, one or more to each element of the structure. composition of several functions may be evaluated concurrently by instantiating several copies on different processors and streamin data through them. This is most often possible when the compositio is the repeated part of an iterative control construct. In anothe mode, each function is instantiated on a separate processor and dat is streamed through in a pipelined fashion. If two or more formal parameters of a function instance have mutually exclusive side effects they may be evaluated concurrently and acheive the same machine state as a sequential evaluation. This will be known as horizontal concurrency.

This paper presents a formalism by which horizontal concurrency can be detected in LISP programs which have global variables and arbitrary side effects. Any execution model which permits some form of explicit sequentialization and permits global variable access and arbitrary side effects can use this method to map functional forms onto their evaluation strategy.

2.0 PRELIMINARIES

There has been much work on data flow analysis in procedura languages. For the most part these methods are predicated on fl graph semantics in which state is affected by assignment and contr flow by simple conditionals. The formalisms used in these previou work [1-6, 9, 11] are modified to fit into the functiona programming environment of LISP. The model used is the definitic of the LISP evaluator, that is, function evaluation in th environment of a run time symbol table, formal-actual paramet bindings, and arbitrary binary tree structures.

We define the global LISP environment in terms of its symbc table and then provide informal semantics of the LISP evaluator The LISP system that is the subject of this analysis is Standar LISP [7] without PROG, ERROR/ERRORSET, vectors, and floating poir numbers.

2.1 The LISP Symbol Table

A function instance is a source language occurrence representing th invocation of a function and the binding of its actual paramete values to its formal parameter names. A function definitio instantiates the application of functions in the environment of th formal-actual parameter bindings.

A LISP identifier is a place holder for three entities. global value can be associated with an identifier and is accessibl by name no matter at what lexic or evaluation level which it occurs

A function may be associated with an identifier and as such is global entity. The semantics we define allow global variabl bindings and function definitions to coexist. The property list i a structure of indicators with associated values. The property lis may also include flags which are indicated by the presence o absence of the identifier by which the flag is known. Flags ma have the same names as indicators but with undefined results. Thes global attributes of an identifier are treated in what follows as attributed identifier which is expressed as:

[indicator : identifier] or [flag : identifier]

The local environment of an extended variable determines its globa binding. If the use is as a global variable, the indicator *BIND will be used. If the identifier names a function, then th indicator *FUNCTION* is used instead. When the property list of a identifier is being accessed, the indicator is the indicator fro the property list.

2.2 A Logic For Property Lists

S is the set of identifiers in the LISP system at any given time. In many systems this corresponds to the OBLIST. The set S′ is defined to be the set {S, *BIND*, *FUNCTION*} to prevent *BIND* and *FUNCTION* from being identifiers in their own right. The possible bindings (that is identifier and indicator) at any given time is the set G* with the set G $\subseteq$ G* being the current set. G* is the set of elements formed by the cross product $S' \times S$ whose elements will be given in the form [a:b], where a is the indicator selected from S' and b is an identifier in S. G* is divided into equivalence classes:

$$\forall a,b \in S', \quad \forall c,d \in S, \ [a:c] \equiv [b:d] \text{ iff } c=d.$$

These equivalence classes correspond to the property lists of identifiers which can include both a function and global value.

A particular identifier g determines an equivalence class $\overline{G} \subseteq G^*$. The operation σ induces a partial ordering on the set $\overline{G}$. When the indicator being accessed by σ is unknown the special indicator $I \notin S'$ is used. σ induces the following relations:

$\forall a,\ b \in S'$:

1. $\emptyset = \sigma(a,\overline{G})$

2. $\sigma(a,\overline{G}) = \sigma(a,\overline{G})$

3. $\sigma(a,\overline{G}) = \sigma(b,\overline{G})$

4. $\sigma(a,\overline{G}) = \sigma(I,\overline{G})$

5. $\sigma(I,\overline{G}) = \sigma(I,\overline{G})$

A minimal representation of $\sigma(x,\overline{G})$ is:

$\forall i,\ 0 \leq i \leq n,\ a[i] \in S^*,$

$\{\sigma(a[0],\overline{G}),\ \sigma(a[1],\overline{G}),\ \ldots,\ \sigma(a[n],\overline{G})\} \ni$

$\forall i,\ 0 \leq i \leq n,$

$\forall j,\ 0 \leq j \leq n,\ i \neq j,\ \sigma(a[i],\overline{G}) \neq \sigma(a[j],\overline{G})$.

Thus any such set containing $\sigma(I,\overline{G})$ will contain only $\sigma(I,\overline{G})$. Othe: sets will contain no redundant elements.

Given a minimal set $B = \{\sigma(b[0],\overline{G}), \ldots, \sigma(b[n],\overline{G})\}$ the unio: of it with a singleton set $A = \emptyset$, or $A=\{\sigma(a,\overline{G})\}$ is defined as:

$A \underline{U} B$ ::=

    <u>if</u> $B = \emptyset$ <u>then</u> A

    <u>else</u> <u>if</u> $A = \sigma(b[0],\overline{G})$ <u>then</u> B

    <u>else</u> <u>if</u> $\exists i,\ 0 \leq i \leq n \ni \sigma(a,\overline{G}) = \sigma(b[i],\overline{G})$ <u>then</u> B

      <u>else</u> $\{\sigma(a,\overline{G}), \sigma(b[0],\overline{G}), \ldots, \sigma(b[n],\overline{G})\}$.

Note that $A \underline{U} B$ is minimal. Set union for two general minimal sets:

$A = \{\sigma(a[0],\overline{G}), \ldots, \sigma(a[m],\overline{G})\}$

$B = \{\sigma(b[0],\overline{G}), \ldots, \sigma(b[n],\overline{G})\}$

is defined:

$A \overline{U} B$ ::= $\{\sigma(a[0],\overline{G}) \underline{U} (\ldots \underline{U} (\{\sigma(a[m],\overline{G}) \underline{U} B) \ldots)\}$

For two sets $H, L \subseteq G^*$ we define set union in terms of equivalence classes of H and L which are H[0]...H[n] and L[0]...L[n] where H[i] and L[i] are corresponding equivalence classes.

$H \cup L$ ::= $\{H[0] \overline{U} L[0], \ldots, H[n] \overline{U} L[n]\}$

Set intersection between a singleton set $A = \emptyset$, or $A=\{\sigma(a,\overline{G})\}$ and B as above is defined as:

$A \underline{\cap} B$ ::=

  if $B = \emptyset$ **then** $\emptyset$

  else if $A = \sigma(b[0],\overline{G})$ **then** A

  else if $\exists\, i,\ 0 \le i \le n,\ \ni\ \sigma(a,\overline{G}) = \sigma(b[i],\overline{G})$ **then** A

    else $\emptyset$.

For the two minimal sets A and B above:

$A \overline{\cap} B$ ::= $\{(\sigma(a[0],\overline{G}) \underline{\cap} B),\ \ldots\ ,\ (\sigma(a[n],\overline{G}) \underline{\cap} B)\}$

And for H and $L \subseteq G^*$ as before:

$H \cap L$ ::= $\{H[0] \overline{\cap} L[0],\ \ldots\ ,\ H[n] \overline{\cap} L[n]\}$

Finally, with $\overline{L}$ being set complement with respect to G^*:

$H - L$ ::= $H \cap \overline{L}$.

To permit fixed indicator, arbitrary identifier access the special set notation [a:I] is defined:

[a:I] ::= $\forall i,\ 0 \le i \le n,\ b[i] \in S,\ \{[a:b[0]],\ \ldots,\ [a:b[n]]\}$

where $n = |S|$. Arbitrary indicator and identifier access is permitted but is handled as a completely non-computable side effect.

2.3 Sequential LISP Semantics.

The following is an informal model of sequential LISP similar to the semantics of Standard LISP [8]. The concurrency detected by the global data flow analysis will preserve the semantics of LISP encapsulated in this definition.

The definitions are predicated on the definitions of local and global scope. The global scope refers at any time to the set G*. There is a set A* which is the current local scope. A A* is the current most local bindings. A can be thought of as the top most frame of the stack of frames A*. The frames are stacked and unstacked by two internal state effectors GLOCAL and UNGLOCAL (discussed with LAMBDA). There is a restriction that:

$\forall a$, [*BIND*:a] $\in$ A*, $\forall b$, [*BIND*:b] $\in$ G*, $a \neq b$.

and

$\forall a$, [*DEFINITION*:a] $\in$ A*, $\forall b$, [*DEFINITION*:b] $\in$ G*, $a \neq b$.

That is, all local variables may not be global variables or functions.

The value associated with an attributed identifer is retrieved by the access function ξ.

For atoms a, b, and f, for i an identifier, for S-expressions s, and for x, y list structures the following are defined:

1. CONSTANTP(a) ::= an item which when evaluated yields itself. This includes numeric values, strings, and function pointers.

2. IDP(a) ::= <u>if</u> a $\in$ S <u>then</u> <u>true</u> <u>else</u> <u>false</u>
   IDP is true if a is an identifier. An identifier is an item which when evaluated gives the value currently bound to it, either the current GLOBAL value or the value from the most recent local parameter binding.

3. CODEP(a) ::= an item which is a link to a function whos internal form is not a LISP S-expression.

4. GLOBALP(a) ::= <u>if</u> [*BIND*,a] ∈ G* <u>then</u> <u>true</u> <u>else</u> <u>false</u>

5. GLOBAL(a) ::= ξ [*BIND*:a]

6. LOCAL(a) ::= <u>if</u> [*BIND*:a] ∈ A <u>then</u> ξ[*BIND*:a]

7. DEFINED(a) ::= <u>if</u> [*FUNCTION*:a] ∈ G* <u>then</u> <u>true</u> <u>else</u> <u>false</u>

8. TYPE(f) ::= for a DEFINED function f, TYPE(f) is the type, EXPR, or FEXPR of the function.

9. DEFINITION(f) ::= the body of the definition of the function whose name is f.

10. EVAL(a) ::= <u>if</u> CONSTANTP(a) <u>then</u> a

```
         else if IDP(a) then
           if GLOBALP(a) then GLOBAL(a)
                         else LOCAL(a);
```

The evaluation of an atom is either that of constants, or retrieval of the global or local value bound to the variable.

11. (EVAL (f . x)) ::=

```
     if ~DEFINED(f) then <ERROR - undefined function>
     else if TYPE(f) = EXPR then APPLY(DEFINITION(f), EVLIS x)
     else if TYPE(f) = FEXPR then APPLY(DEFINITION(f), LIST(x));
```

The evaluation of a function invocation is by applying the function definition to the evaluated list of arguments (an EXPR) or to the unevaluated arguments collected into a list and bound to the single parameter of the FEXPR.

12. EVLIS(NIL) ::= NIL

13. EVLIS(x . y) ::= EVAL x . EVLIS y

14. APPLY(f, x) ::=

```
if IDP(f) then
  if not DEFINED(f) then
    {ERROR - undefined function}
  else if TYPE(f) = EXPR then
    APPLY(DEFINITION(f), x)
  else {ERROR - Can't be evaluated by APPLY}
```

If the function to be applied is a defined function, apply its definition to its arguments.

15. APPLY(((LAMBDA . x) . s), y) ::=

i) PUSH(A). Stack the current local environment onto A*.

ii) A = {[*BIND*:x[0]], ..., [*BIND*:x[m]]}. ξ[*BIND*:x[0]] = y[0], ..., ξ[*BIND*:x[m]] = y[m]. Bind the values in y to the corresponding local variable names in x.

iii) Compute EVAL(s).

iv) A = POP(A*). Redefine the current set of local variables to its previous contour.

v) APPLY(((LAMBDA . x) . s), y) = value from step iii.

LAMBDA functions cause the instantiation of LOCAL type variables for the lexical scope of the LAMBDA expression (that is, s).

The current LOCAL binding is not accessible outside of t lexical scope of the LAMBDA as it is in many LISP interpreter The semantics of local variables are that used by most LI compilers.

2.4 Variables And Their Use.

The following terms are defined assuming the sequenti interpretation of evaluation of function instances.

Definitions: An *extended variable* in a function F is a fr variable in F. An *extended occurrence* of a variable in function instance f is a free occurrence of the variable in An extended variable occurring in a function F is a *sour* variable iff the evaluation of F is not affected by the initi value of the variable when the function is invoked. An extend variable occurring in a function F is an *access* variable iff fo no possible execution of F is the variable rebound. An extende variable occurring in a function F is a *changed* variable if ther exists an execution of F in which the evaluation of F depends c its value and then modifies its value. A *hard* function has nonlocal effect which cannot be determined by our method excep by evaluation in its environment. A *soft* function has no suc effect.

The following sets of variables of a given function F are defined:

SF - the set of source variables

AF - the set of access variables

CF - the set of changed variables

As a consequence, SF, AF, and CF are pairwise disjoint for any function F. There is a boolean value associated with each function:

HF - is true if function F is hard

An _R quadruple_, RF, is defined as the ordered quadruple (SF, AF, CF, HF). Also defined are the standard quadruples Rø = (ø, ø, ø, false) and Rt = (ø, ø, ø, true). It is assumed that for any function F, RF is known.

2.5 Data Flow In Single Rooted Directed Trees.

A set of operations on R quadruples are defined with which to determine whether or not two functional instances can be evaluated concurrently without affecting their sequential semantics. The two operators <--> and --> are the basic data flow equations for LISP.

The operator <--> maps two function instance R quadruples into a resultant R quadruple representing the semantics of the sequential evaluation. For two functional instances a and b:

Ra <--> Rb = (Sc, Ac, Cc, Hc), where:

$$Sc = (Sa \cup Sb) - (Ca \cup Cb) - (Aa \cup Ab)$$

$$Ac = (Aa \cup Ab) - (Sa \cup Sb) - (Ca \cup Cb)$$

$$Cc = (Ca \cup Cb) \cup ((Sa \cup Sb) \cap (Aa \cup Ab))$$

$$Hc = Ha \vee Hb$$

The <--> operator defines what happens to extended variables in th sequential evaluation of a and b. The set of source variables in R is the union of the source variables in Ra and Rb less any whic appear as changed or access variables in either. The set of acces variables in Rc is the union of the access variables in Ra and R less any that appear as changed or source variables. The set o changed variables in Rc is the union of the set of changed variable in Ra and Rb and any that appear as both source and acces variables. If either a or b has a hard effect, Rc will also.

Note that the symmetry of set operations implies that:

Ra <--> Rb = Rb <--> Ra

Consequently no matter what order the actual parameters are evaluated in, the R quadruple of the result is the same. It is most often the case that the sequential semantics dictate a "first to last" evaluation and thus the (Rb <--> Ra) case will never occur (if this sequential order can be determined). The restriction of <--> to sequential order is the --> operator defined over two R quadruples Ra and Rb such that Ra --> Rb is defined as the R quadruple of evaluating 'a' before 'b'.

Ra --> Rb = (Sc, Ac, Cc, Hc) where:

$$Sc = Sa \cup (Sb - (Aa \cup Ca))$$

$$Ac = (Aa - (Sb \cup Cb)) \cup (Ab - (Sa \cup Ca))$$

$$Cc = Ca \cup ((Cb - Sa) \cup (Aa \cap Sb))$$

$$Hc = Ha \vee Hb$$

The --> operator removes variables from the changed class that a first set and the accessed to the source class. Variables which a accessed, then set, are removed from the access class and added the changed class.

The defined evaluation of a LISP EXPR function instar (f a[0] a[1] ... a[n]) is left to right. That is, first a[0] evaluated, then a[1], and so on. To compute the R quadruple of t instance of F, the --> operator is applied across the arguments of by the RSPREAD function to create an R quadruple for the instanc For the function instance f, RSPREAD is defined:

1. RSPREAD(f . NIL) ::= Rø

2. RSPREAD(f . (x . y)) ::= RQUADRUPLE(x) --> RSPREAD(y)

The construction function RQUADRUPLE used above maps functi composition to R quadruples. Assume that RSEMANTIC(f) a RSEMANTICP(f) are known for all functions in the system. For function instance f and atom a:

1. RQUADRUPLE(a) ::= <u>if</u> CONSTANTP(a) <u>then</u> Rø
   <u>else</u> (ø, {[*BIND*:a]}, ø, false)

2. RQUADRUPLE(f . x) ::=
   <u>if</u> RSEMANTICP(f) <u>then</u> APPLY(RSEMANTIC(f), x)
   <u>else</u> RSPREAD(x) --> Rf

APPLY causes evaluation of evaluation of a function with arguments. In this case, the semantic definition associated with a function which does not have its arguments evaluated is applied to the argument list. Note that the last line of the definition of RQUADRUPLE defines function composition as a sequential process. That is, the actual parameters are evaluated first, then the function is applied to their values.

The RSEMANTIC function used in the definition of RQUADRUPLE is a selector of special forms in which the sequential evaluation is not strictly left to right, or ones which are primitive and have side effects. It returns a semantic function used to define the evaluation sequence in terms of applications of --> and <--> which is applied to the argument list of the instance. Some of these semantic functions will appear later.

Two relations are introduced defining the order of evaluation of functional arguments which preserves the sequential semantics of LISP. For two functional instances "a" and "b" in a sentential form (f ... a ... b ...), let:

A = RQUADRUPLE(a) and B = RQUADRUPLE(b)

If "a" is evaluated before "b" in the sequential mode then c
of the following is true:

1. $(Sa \cup (Sb \cup Ab \cup Cb) \neq \emptyset) \vee$

   $(Aa \cup (Sb \cup Cb) \neq \emptyset) \vee$

   $(Ca \cap (Sb \cup Ab \cup Cb) \neq \emptyset) \vee Ha$

   If true we say "a << b", "a" must be evaluated before "
   because the effect of "a" affects the evaluation of "b".

2. $(Sa \cap (Sb \cup Ab \cup Cb) = \emptyset) \wedge$

   $(Aa \cap (Sb \cup Cb) = \emptyset) \wedge$

   $(Ca \cap (Sb \cup Ab \cup Cb) = \emptyset) \wedge \neg Ha \wedge \neg Hb$

   If true we say "a == b", "a" and "b" may be evaluat
   concurrently because the evaluation of "a" does not affect t
   evaluation of "b" and the evaluation of "b" does not affect t
   evaluation of "a".

When the sequential order of evaluation cannot be determined th
one of the following is true:

1. a << b or a >> b then sequential evaluation is indicated
   preserve the semantics.

2. a == b, "a" and "b" can be evaluated concurrently becau
   neither can affect the other.

2.6 Semantics Of Modifiers And Access Functions.

A class of functions called *modifiers* have nonlocal effec upon their environment. If the location of this effect can exactly computed before the evaluation of the function instance, t effect is known. If the location cannot be computed except by t evaluation of the instance in its environment, the effect is n known. These two possibilities are known as *soft modifiers* and *ha modifiers* respectively.

The computation of the R quadruple for each modifier occurren is performed by a semantic function associated with it. T semantic functions take as parameters the formal parameters of t function being modeled. A few semantic functions associated wi modifiers are presented here.

RSEMANTIC(SET a b) ::= Rt

RSEMANTIC(SET (QUOTE a) b) ::=

({[*BIND*:a]}, ∅, ∅, *false*) --> RQUADRUPLE(b);

If the argument of SET is a quoted variable name then the effe of SET is known (a soft modifier). Otherwise, SET is a ha modifier with the R quadruple indicating a non-computable si effect.

RSEMANTIC(SETQ a b) ::=

({[*BIND*:a]}, ∅, ∅, *false*) --> RQUADRUPLE(b);

Here "a" is a source variable combined by --> with the quadruple of the second argument "b".

RSEMANTIC(RPLACA a b) ::= Rt

RSEMANTIC(RPLACD a b) ::= Rt

Because of aliasing problems, RPLACA, RPLACD and functions whi use them are always hard functions and instances. Since a ha function forces complete sequentialization, the R quadruple "b" need not be computed for this instance. The evaluation "b" may have available concurrency, but it must all be complet before the RPLACA or RPLACD is started.

RSEMANTIC(PUT (QUOTE a) (QUOTE b) c) ::=

({[b:a]}, ∅, ∅,<u>false</u>) --> RQUADRUPLE(c);

RSEMANTIC(PUT (QUOTE a) b c) ::=

({[I:a]}, ∅, ∅, <u>false</u>) --> RQUADRUPLE(b) --> RQUADRUPLE(c);

RSEMANTIC(PUT a (QUOTE b) c) ::=

(RQUADRUPLE(a) --> ({[b:I]}, ∅, ∅, <u>false</u>)) --> RQUADRUPLE(c)

RSEMANTIC(PUT a b c) ::= Rt

There are four possible cases for PUT. In the first, both t identifier being modified, and its indicator are precisely kno and the R quadruple has the [b:a] pair as a source variabl This is combined by the action of --> with what ever happens the evaluation in "c". In the second case, we know that "a" the identifier being accessed, but the indicator non-computable giving rise to the [I:a] form. In the third cas the identifier is non-computable, but the indicator is know consequently the [I:b] form. In the final case, nothing can computed about the instance and a complete sequentialization forced.

The definition of the data flow semantics of PUT is prototype for the very similar functions PUTD, FLAG, and REMFLA

RSEMANTIC(DE f name args body) ::=

({[*FUNCTION*:f]}, ø, ø, false)

RSEMANTIC(DF f name args body) ::=

({[*FUNCTION*:f]}, ø, ø, false)

DE and DF are FEXPR's, and they don't evaluate their argument The only effect is to define the function and the R quadrup need not be computed for each of the arguments.

RSEMANTIC(REMPROP (QUOTE a) (QUOTE b)) ::=

(ø, ø,{[b:a]}, false)

RSEMANTIC(REMPROP (QUOTE a) b) ::=

(ø, ø,{[I:a]}, false)

RSEMANTIC(REMPROP a (QUOTE b)) ::=

RQUADRUPLE(a) --> (ø, ø,{[b:I]}, false)

RSEMANTIC(REMPROP a b) ::= Rt

REMPROP removes the indicator "b" and its value from the prope list of "a". It returns the value associated with the indica "b" so "a" becomes a changed variable rather than a source. function REMD for removing functions is similar. The funct REMFLAG is like PUT because it does not reference the fl before removing them.

RSEMANTIC(GET (QUOTE a) (QUOTE b)) ::=

(ø, {[b:a]}, ø, false)

RSEMANTIC(GET (QUOTE a) b) ::=

(ø, {[I:a]}, ø, false) --> RQUADRUPLE(b)

RSEMANTIC(GET a (QUOTE b)) ::=

RQUADRUPLE(a) --> (ø, {[b:I]}, ø, false)

RSEMANTIC(GET a b) ::= Rt

GET simply accesses a portion of the property list of identifier. In most systems GET is not permitted to access t global binding or the function definition. In this case similar function is defined for these two special cases (t functions GLOBAL, and DEFINITION were used in the exposition EVAL APPLY earlier). I have chosen to incorporate the semanti of global and local variable access in the RQUADRUPLE functi rather than at this level. The reasoning is that variable acce in any form is important enough to hide the access functions fr the user.

The functions GETD, GLOBALP, and FLAGP are very similar GET and are not defined here.

2.7 Other Functions

There are a large number of primitve EXPR type functions which fo the basic processing capabilities of the system. For the most pa they have R quadruples of Rø. The few which do not or are suprisi are presented here.

RSEMANTIC(COMPRESS x) ::= Rø

COMPRESS does little more than create atoms of various sort When the attributes of these atoms are accessed by oth functions the identifiers are added to the R quadruple, usual: in the form of Ṙt.

RSEMANTIC(EXPLODE x) ::= R∅

EXPLODE does not reference the quantity it is creating nor tl characters of the list it creates.

RSEMANTIC(GENSYM) ::=

({[I:gensym]} , ∅, {[*BIND*:gensym-counter]}, <u>false</u>)

GENSYM both changes a global variable which contains a counter cause creation of unique symbols. Likewise it is a source f the property list of the created symbol.

RSEMANTIC(INTERN x) ::= ({[I:x]}, ∅, ∅, <u>false</u>)

INTERN augments the current set of symbols S and also destro any property list associated with x. Thus it is a source for a of x.

RSEMANTIC(EVAL x) ::= RQUADRUPLE(x)

The EVAL function performs no modifications on the global stat rather it relies on its argument for this effect. The same true for APPLY.

2.8 Recursive Functions

The problem of data flow analysis of recursive functions consists two subproblems: functions which are directly recursive, and tho which are indirectly recursive. The indirect recursive functi problem will not be solved here. Rather the ad hoc solution o having the user directly specify the R quadruple of at least on element of the recursive chain will be adopted.

Directly recursive functions can be analyzed by recognizin that they always involve the computation of Rf -> Rf for th recursive function f. It is easy to show that Rf -> Rf ≡ Rf b appropriate substitutions into the -> equation. Likewise it ca also be shown that Rø -> Rf ≡ Rf. Consequently, directly recursiv calls in functions can be treated as Ro instances without affectin the outcome of the analysis. Since the implementation of availabl concurrency happens after the computation of Rf, Rf for th recursive instance will be known.

2.9 FEXPR's

Most LISP's permit the definition of functions with an arbitrar number of arguments in which the order of evaluation is defined b the function itself. For arbitrary functions the implementor mus supply an appropriate RSEMANTIC function as in general, the proble of machine definition of such a function appears to be ver difficult.

RSEMANTIC functions compute the "worst case" instance o evaluation, that is, the one with the least amount of concurrenc that evaluates the most arguments. We define some of the mos important LISP functions here.

```
RSEMANTIC(AND NIL) ::= Rø
RSEMANTIC(AND (x . y)) ::=
   RQUADRUPLE(x) --> RQUADRUPLE(AND y)
```

The AND function and OR, PROGN, MAX, MIN, TIMES, PLUS and othe forms will in the "worst case" evaluate all their argument sequentially. To take advantage of a non-deterministic MAX, ar MIN would require redefining their semantics because of thei ability to deal with mixed mode arguments (integer and floatir point).

```
RSEMANTIC(COND NIL) ::= Rø
RSEMANTIC(COND ((a . (c)) . x) ::=
   (RQUADRUPLE(a) --> RQUADRUPLE(c)) -->
      RQUADRUPLE(COND x)
```

The data flow semantics of COND are very similar to AND. Ther is available concurrency in individual antecedent-consequer elements but none in the structure as a whole.

2.10 Local References And Effects

Functional forms that introduce local variables effect the detectic of available concurrency within their scope by temporarily modifyir the local environment. The computation of R quadruples for thes nested forms must include this contour information. Variables c this sort can be reassigned in the current environment with SET ar SETQ temporarily introducing new binding values. Consequently thes variables will be treated as temporary global variables, glocals, a acronym for global and local. The variables are tagged as such b

the effector functions GLOCAL and UNGLOCAL. The GLOCAL functi
pushes the crrent A into A* and creates a new set of local variabl
v. The UNGLOCAL function removes these and returns the previous s
of locals from A*.

LAMBDA forms permit local modification of actual paramet
values. This modification is in effect only in the lexical scope
the LAMBDA form. To describe this effect requires an operation
description of the semantic properties of LAMBDA.

```
RSEMANTIC(LAMBDA v b) ::=
   1.  GLOCAL v;
   2.  x := RQUADRUPLE(b) -
      ({[*BIND*:v[0]], ... , [*BIND*:v[n]]},
       {[*BIND*:v[0]], ... , [*BIND*,v[n]]},
       {[*BIND*:v[0]], ... , [*BIND*,v[n]]}, false)
   3.  UNGLOCAL v;
```

The RSEMANTIC function for LAMBDA has the value of the second form
The environment is affected after this value is computed. Note tha
this definition of LAMBDA permits local modification of forma
parameter values with the SET and SETQ functions.

3.0 CONCLUSIONS

Potential horizontal concurrency in simple LISP functions i
detected using global data flow analysis techniques. Application o
this knowledge to the creation of concurrently executable program
is dealt with in [7].

The analysis has been simplified by two assumptions. T analysis covers LISP functions whose flow graphs are single direct rooted trees. Secondly, it is assumed that the R quadruples for a functions are known before they are needed. The first assumptio can be removed by employing path expression analysis [5, 10] permit the analysis of LISP PROG forms and other forms of flo control. The second can be removed only if the incremental propert of LISP systems is remcved and blocks of code can be treated a single entities.

List of References

1. Aho, A. V., Ullman, J. D. *Principles of Compiler Desig* Addison-Wesley Publishing Company, Reading, Massachusetts, 197 429-438.

2. Allen, F. E. Interprocedural data flow analysis. *IFIP 7* North-Holland Publishing Company, Amsterdam, 1974, 398-402.

3. Allen, F. E., Cocke, J. A program data flow analys procedure. *Commun. ACM 19*, 3 (March 1976), 137-147.

4. Barth, J. M. A practical interprocedural data flow analys algorithm. *Commun. ACM 21*, 9 (September 1978), 724-736.

5. Graham, S. L., Wegman, M. A fast and usually linear algorit for global flow analysis. *J. ACM 23*, 1 (January 1976 172-192.

6. Kam, J. B., Ullman, J. D. Global data flow analysis a iterative algorithms. *J. ACM 23*, 1 (January 1976), 158-171.

7. Marti, J. Compilation techniques for a control-flow concurre LISP System. *Conference Record of the 1980 LISP Conferenc* 1980, 203-207.

8. Marti, J., Hearn, A. C., Griss, M. L., Griss, C. Standa LISP report. *SIGPLAN Notices 14*, 10 (October 1979), 48-68.

9. Rosen, B. K. High-level data flow analysis. *Commun. ACM 2* 10 (October 1977), 712-724.

10. Tarjan, R. E. Fast algorithms for solving path problems. ACM 28, 3 (July 1981), 594-614.

11. Weihl, W. E. Interprocedural data flow analysis in presence of pointers, procedure variables, and label valu Proceedings of 7th Annual Symposium on Principles of Programm Languages, 1980, 83-94.