MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence
Memo. No. 151

January 1968.

Functional Abstraction in LISP and PLANNER

Carl Hewitt.

Presented here is part of the graduate work that I am doing in the much broader area of protocol analysis (see A.I. memo 137). The goal of functional abstraction is to find a procedure that satisfies a given set of fragmentary protocols.0 Thus functional abstraction is the inverse operation to taking a set of protocols of a routine. The basic technique in functional abstraction (which we shall call IMAGE) is to find a minimal homomorphic image of the set of fragmentary protocols. It is interesting to note that the technique of finding a minimal homomorphic image is the same one used to compute the schematized goal tree in A. I. memo 137 . We define (a < b) to mean that a is erased and b is written in its place. We shall use (a : b) to mean that the value of b is a. Consider the folowing protocol for a function f:

```
arguments of f = (3)
(() < 1)
(() : (pred  3))
(1 < (3 : (fcn 3 1)))
(3 < (2 : (fcn 3)))
(() : (pred 2))
(3 < (6 : (fcn 2 3)))
(2 < (1 (fcn 2))
```

```
(() < (pred 1))
(6 < (6 : (fcn 1 6)))
(1 < (0 : (fcn 1)))
(T : (pred 0))
value of f = 6
```

If IMAGE knows about the functions zerop, times, and sub1 then it will find the following minimal homomorphic image for the above protocol:

```
(defprop f
(lambda (arg1) (prog (temp1)
    (setq temp1 1)
label1 (cond ((zerop arg1) (return temp1)))
    (setq temp1 (times arg1 temp1))
    (setq arg1 (sub1 arg1))
    (go label1 ))) expr)
```

The next example makes use of MATCHLESS (see A.I. memo 137). We use dashes to mark the ends of a fragment of a list. The atoms *var*, =var=, and var have the effect that they put the pattern variable which follows them in the var mode.

```
arguments of f = ((- -))
(T : (pred (- -) () ))
value of f = T

arguments of f = ((-a-))
```

```
(() : (pred (-a-) () ))
((-a-): (fcn (( () < a) (-a- < - -)) (-a-) ) )
(T : (pred a a))
(T : (pred (- -) () ))
value of f = ()

arguments of f = ((-b-))
( () : (pred (-b-) ()))
( (-b-) : (fcn (( () < b) ( -b- < - -) ) (-b-) ))
(() : (pred b a))
(T : (pred (- -) ()))
value of f = ()

arguments of f = ((-a b-))
(() : (pred (-a b-) () ))
((-a b-)  : (fcn (( () < a) (-a b- < -b-)) (-a b-) ))
(T : (pred a a ))
( () : (pred (-b-) () ))
( (-b-) : (fcn ( (a < b) ( -b- < - -)) (-b-) ))
( () < (pred b a))
( T : (pred ( - -) () ))
value of f = T

In this case IMAGE gives
(defprop f
(mlambda (*arg1) (prog (=temp1)
label1 (cond ((assign? (*arg1) ()) (return T)))
    (assign ((=var= =temp1) (*var* *arg1)) (*arg1))
```

```
    (cond ((assign? =temp1 a) (go label2)) )
label2 (cond ((assign? (*arg1) () ) (return ()))
    (assign ((=var= =temp1) (*var* *arg1)) (*arg1))
    (cond ((assign? =temp1 a) (go label1)))
    (go label1))) fexpr)
```

In the above mlambda causes (*arg1) to be assigned to the list of arguments of f. The generalized assignment statements of MATCHLESS are "assign" and "assign?".

For some purposes in functional abstraction, minimal homomorphic images are not ideal representatives of functions. The predicate sequence of an element of a protocol is defined to be the sequence of values taken on by the predicates of the protocol from the beginning of the protocol to the element. If h is a homomorphism then we say that h is a branch preserving homomorphism if h(a) = h(b) implies that the predicate sequence of a is an initial segment of the predicate sequence of b or vice-versa. We shall call a minimal branch preserving homomorphic image a bimage. For example a bimage for the above protocol is

```
(defprop f
(mlambda (*arg1) (prog (=temp1)
label1 (cond ((assign? (*arg1) ()) (return T)))
    (assign ((=var= =temp1) (*var* *arg1)) (*arg1))
    (cond ((assign? =temp1 a) (go label3)))
label2 (cond ((assign? (*arg1) ()) (return ()))
```

```
    (assign ((=var= =temp2) (*var* *arg1)) (*arg1))
    (cond ((assign? =temp1 a) (go label1)))
    (go label1)
label3 (cond ((assign? (*arg1) ()) (return ())))
    (assign ((=var= =temp1) (*var* *arg1)) (*arg1))
    (cond ((assign? =temp1 a) (go label1)))
    (go label1))) fexpr)
```

Bimages have certain advantages and disadvantages compared to images. As illustrated above images are almost always smaller. On the other hand bimages take less time to compute and are easier to modify in case of error. To correct a bimage it is necessary only to break some of the feed back loops and insert more code.

One of the most important applications of functional abstraction is for the problem of making the robot do some task after it has seen several examples of how the task is done. Suppose that we want the robot to take all the blocks out of a box and stack them in a tower at the place P. We place three cubes in the box and then take them out one by one as the robot watches and stack them at P. After observing the example we would like for the machine to write out a protocol something like the following:

```
(in block1 box)
(in block2 box)
(in block3 box)
(erase (in block1 box))
```

```
(assert (above block1 P))
(assert (on block1 table))
(erase (in block2 box))
(assert (above block2 P))
(assert (on block2 block1))
(erase (in block3 box))
(assert (above block3 P))
(assert (on block3 block2))
(finished)
```

If we did the same thing again only with two blocks instead of three at the place Q we would expect the folowing protocol:

```
(in block4 box)
(in block5 box)
(erase (in block4 box))
(assert (above block4 Q))
(assert (on block4 table))
(erase (in block5 box))
(assert (above block5 Q))
(assert (on block5 block4))
(finished)
```

We compute the following image of the above protocols:

```
(thprog ($var1 $var2 $var3)
    (consequent (tower at $var1))
```

```
      (thcond ((not (erase (in $var2 box))) (finished)))
      (assert (above $var2 $var1))
      (assert (on $var2 table))
 label1 (assign ($var$ $var3) $var2)
      (thcond ((not (erase (in ($var$ $var2) box))
(finished)))
      (assert (above $var2 $var1))
      (assert (on $var2 $var3))
      (go label1))
```

We can convert the above image into a function that builds towers by interpolating the commands necessary to carry out the state changes:

```
 (thprog ($var1 $var2 $var3)
      (consequent (tower at $var1))
      (thcond ((not (erase (in $var2 box))) (finished)))
      (pickup $var2)
      (assert (above $var2 $var1))
      (move-to $var1)
      (assert (on $var2 table))
      (lower-arm-to-contact)
      (release-cube)
      (check-scene (on $var2 table))
 label1 (assign ($var$ $var3) $var2)
      (thcond ((not (erase (in ($var$ $var2) box))
(finished)))
```

```
(pickup $var2)
(assert (above $var2 $var1))
(move-to $var1)
(assert (on $var2 $var3))
(lower-arm-to-contact)
(release-cube)
(check-scene (on $var2 $var3))
(go label1))
```

IMAGE is an important technique for functional abstraction for a variety of reasons. IMAGE is self applicable in the sense that given IMAGE, we can have the machine abstract IMAGE from fragmentary protocols of IMAGE. PLANNER is well suited to the task of constructing images since it permits small mistakes to be made in the construction without causing fatal errors. It is useful to allow declarations to be included with the set of protocols to be analyzed. In the case of the factorial function, it is very useful to know that in the set of protocols arithmetic is assumed to be done base 8. Of course once an image has been constructed it can used to build further images. Thus IMAGE shows one way in which previous results can be incorporated to build up more complicated functions. The protocol given above whose functional abstraction happens to be the factorial function is a good example of how it can be very misleading to say that computers can do only what they are precisely told how to do.