Another Stab at Monads

AA

April 29, 1991

1 The Idea

A natural problem that arises in programming is to take a program that does something and produce another program that does something "like" the first. Programming languages provide various mechanisms for making useful modifications to the functionality of a program with minimal effort. A good example is the mechanism of inheritance in object-oriented languages. Another example is (some uses of) state in imperative languages. For instance, usually by changing only one procedure, a counter can be added that keeps track of some interesting value. To do this in a functional language, one is forced to modify not just one function, but every function that depends, directly or indirectly, on that function as well.

In functional languages, the only general solution proposed to date is Wadler's monads. The idea is to add new syntax to the language that specify how to interpret the text of the program; whether it is to be interpreted normally, or in some special monad. Monads are simple enough, it is argued, that little semantic clarity is lost in exchange for a powerful new notation.

The key here is that ***new syntax*** is used to change the interpretation of the language. Nnew syntax required is required because, in a language based on the lambda calculus (an object-level language), the meaning of lambda abstraction is fixed. There is no way for a user to define a new interpretation within the language. FL is in a better position, however, since the only building blocks with fixed semantics are notated sequences $\langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle$ and function applications $\mathtt{f}{:}\mathtt{x}$. All functions (well, almost all) can be redefined.

This observation leads to the following proposal: instead of adding a new feature to FL, the various monads of interest can be implemented directly by an extended environment of primitive functions. The effect of a new

interpretation of the program is achieved by using a different environment of primitive functions.

This idea has some problems which are mentioned after the examples.

2 New Primitives

We define a new datatype "state". A state is a value paired with a store. The primitives are redefined to have the effect of threading a state through a program. Using this environment modifies programs to map states to states. Some new functions are made available that inspect and modify the store. For convenience, each primitive is overloaded with its standard definition, which is used if it is not applied to a state.

The following are a few illustrative examples of the primitives extended to use states. Most primitives are straightforward, like **isfunc** and ∘ . A few are more difficult, such as **C** . Other involved primitives, which are not presented below, include **cons** and **pcons** .

```
{
nrdef isfunc ≡ isstate → mkstate ∘ [isfunc ∘ val, store]; isfunc

nrdef ∘ ≡ isstate → comp1; ∘

nrdef C ≡ isstate → C1; C
where
{
def comp1 ≡ seqof:isfunc ∘ val →
    mkstate ∘ [ ∘ ∘val, store];
    sigstate:[" ∘ ","arg1",id]

def C1 ≡ isfunc ∘ val → GC:2:C2; sigstate:["C","arg1",id]

def C2 ← [[f.isfunc, a.]] ≡
    isstate ∘ a →
        mkstate ∘ [(GC:3:C3 ∘ f):'(val ∘ a), store ∘ a];
        C ∘ f:'val ∘ a

def C3 ← [[f.isfunc, a1., a2]] ≡
    isstate ∘ a2 →
        f:'mkstate ∘ [[a1, val ∘ a2], store ∘ s2];
        f:'[a1, a2]
```

```
}
uses
{
type state ≡ [[val., store.isstore]]

nrdef lookup ≡ C:lookup

    where{def lookup ← [[i.isstring, s.isstate]] ≡
            mkstate ∘ [find ∘ [i, store ∘ s], store ∘ s]}

nrdef set ≡ C:set

    where{def set ← [[i.isstring, s.isstate]] ≡
            mkstate[val ∘ s, save ∘ [[i, val ∘ s], store ∘ s]]}

nrdef sigstate ≡ C:sigstate

    where{def sigstate ← [[f.isfunc, s.isstate]] ≡
            signal ∘ mkstate ∘ [f:'val ∘ s, store ∘ s]}

}}
```

3 An Example

Below is a (very poor) program that tests membership of an object in a list. The goal is to change the program so that it returns a pair ⟨**bool**, **int**⟩ , where the second component of the pair is a count of the number of times the object appears in the list.

```
def member ← [[x.isobj, s.isseq]] ≡ and ∘ member1

def member1 ← [[x.isobj, s.isseq]] ≡
    isnull ∘ s → []; al ∘ [x = s1 ∘ s, member1 ∘ [x, tl ∘ s]]
```

In the new program, the only changes are to member1 and the addition of some code around the entire function. The environment of state-transforming primitives is used.

```
def newmember ← [[x.isobj, s.isseq]] ≡
    [val, val ∘ lookup:"count"] ∘ member

    uses{
```

```
    /* ... old code .. */
    def member1 ← [[x.isobj, s.isseq]] ≡
        isnull ∘ s → []; al ∘ [addcount ∘ (x = s1 ∘ s), member1 ∘ [x, tl ∘ s]]

    def addcount ← isbool ≡
        id → tt ∘ set:"count" ∘ (lookup:"count" + ~1); id
    }

where lib("SPF")
```

4 Problems

Unfortunately, there is more to the story. Not everything (not even all functions) in FL can be redefined. Some functions have special syntax whose meaning survives redefinition (e.g., pcons). We have the same problem with lambda abstraction as any other language. There may be other dark corners that I haven't thought of.

Another problem is that it is tedious to define the extended environments of primitives (although this is done only once). Many of the extensions are the same for different functions. A more powerful redefinition facility (such as "extend") would help.