Giuseppe Maggiore Michele Bugliesi Universit Ca' Foscari – Venice

1 Introduction

In this paper we present a technique that aims towards supporting many different programming paradigms and techniques with a clear and intuitive syntax (and no particular knowledge requirements) for the developer. We also wish to make it possible to easily switch between one paradigm and the other without modifying the client code: at the "flipping of a switch" the actual execution of a developer code could radically change to support different features *without any modification whatsoever.* The kind of techniques we wish to support as a "runtime backend" for client programs is quite a large set; among these various techniques we discuss some of our favorites: transactional systems, concurrent/distributed systems and reactive programs. We absolutely wish to avoid creating a new language or to extend the syntax of an existing one; instead we try to push the expressivity o the highly flexible Haskell programming language so that it supports our desired constructs. As a starting point we build a framework for working with mutable objects; this system must (of course) be type-safe and also as generic as possible: for this reason we will use meta-programming (type predicates and type functions) a lot. Our object system heavily uses heterogeneous lists as a basis for a customized type-safe heap. We define references to values inside our heap to offer support for mutable operations such as assignment and in-place modification; our heap requires some notion of subtyping to allow references that expect a smaller heap to work wherever a larger heap is active. Whenever a reference represents a value based on our heterogeneous lists then we give a selection operator that returns a reference to a field inside the references heterogeneous list; thanks to this selection operator we now have built a framework for handling mutable records. We extend our mutable records with support for methods and method selection and invocation; since our methods are not invoked directly but through our customized selection operator we can control not only the automated passing of the *this* value to each method as its first parameter, but also the propagation of effects into *this* after the method invocation. Some work is dedicated to overloading the same selection operator so as to not having to distinguish between selecting a field and a method. We add support for inheritance so that the fields and methods of the inherited object can be accessed from the inheriting object in a fully transparent way, and we also add an explicit casting operator to convert a reference to an object to a reference to the *base* (inherited) object.

We pay attention to working on two distinct levels. On one hand we define all our constructs as very generic type predicates and type functions, thereby giving the general shape of our operators. In the most decoupled way possible we instance these functions and predicates with heterogeneous lists, a concrete representation for references, record labels, methods, etc. At the interface level we also define a way to write our computations as statements in the state monad. This allows a developer to write code that uses very general objects, without caring about the plumbing involved whenever the various assignment, selection, method invocation etc. operations occur; by instancing our generic operators multiple times we get the bonus of executing client code with a completely different runtime behavior than simple stateful objects. We will show a few examples of how we can do so and what kind of constructs we may implement. As a first example we will show how we could add three simple operators (*beginT*, *commitT* and *discardT*) to enable transactional code. Then we will show how we could support reactive programming by modifying the behavior of the binding and selection operators, storing and passing on appropriate callbacks that signal the need for propagating changes in the operators to the results of computations. Finally we will discuss how we could make our objects distributed or concurrent, either with shared memory or outright message-passing: by storing a process ID or machine ID inside each reference we could track to which process the current reference belongs to so that whenever we try and access the value represented by such reference we can check whether or not this reference is local and if it is not we will use some synchronization/message-passing scheme to access it.

Our last step will compare our approach to various, more specific approaches for supporting mutable objects, reactive computations, concurrency, etc. on top of pure functional languages.

Note on syntax: we will use an ambiguous set of operators with some conventions. The ambiguity will always be clearly resolved by the context, since some operators will both be present at the type level and at the value level, but there is never any confusion as to whether we are manipulating types or values. Also, we will use a Haskell-like syntax:

1. $x \ = \ v$ for binding
2. $let \ x \ = \ v_1 \ in \ \ v_2$ for binding
3. $T = t$ for defining types
4. $F \ : (* \ \rightarrow \ )^n *$ for giving the kind of a type function
5. $F \ T_1 \ldots \ \ T_n = T_{n+1}$ for specifying the returned value of a type function with certain input parameters
6. $P \ v_1 \ldots \ v_n$ for defining a type predicate
7. $P_1 \ v_{11} \ldots \ v_{1m} \Rightarrow P_1 \ v_{21} \ldots v_{2m}$ for defining a type predicate that requires another type predicate
8. $P \ T_1 \ldots \ T_n$ for instancing a type predicate

Rather than use the full Haskell syntax, we will try and adhere to a more standard syntax. For example, the fact that x has type T we will write

x : T

Rather than the peculiar

```
x::T
```

Used in Haskell programs. Also, when defining a type predicate and the operators it enables we will just say:

```
P  v₁ ... vₙ
f₁  :  T₁
    ⋮
fₘ  :  Tₘ
```

Rather than the Haskell

```
class  P  v₁ ... vₙ  where
  f₁  :  T₁
      ⋮
  fₘ  :  Tₘ
```

2 State Monad

We build a system for performing stateful computation. This kind of computations can be expressed very cleanly with the state monad, which essentially allows us to express the denotational semantics of our statements and which automatically concatenates the semantics of those statements that must be executed sequentially. An ulterior incentive towards using monads is that as language constructs monads have seen an impressive support in many languages, starting from Haskell and ranging to F# and C#; this is probably related to the expressive power of monads and to the way they allow extending a language cleanly and seamlessly, especially if supported by proper syntactic sugar. A statement that evaluates to a value α with a state s in the state monad has type

```
ST  s  α  =  s  ->  (α×s)
```

A monad is a triplet of:

1. a type constructor M that denotes the type of our monads
2. a *unit* operator with type $\alpha \to M\ \alpha$ that generates a monad encapsulating a value
3. a *bind* operator with type $M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta$ that "concatenates" two monads into one

For the state monad we have:

```
M = ST s
unit x = ST(λs.(x,s))
bind (ST m) k=ST(λs.
                  let (x,s')=m s in
                  let (ST m')=k x in
                  m' s')
```

We also add an evaluation operation for values of the state monad:

```
runST : ST s α -> s -> α
runST m s = fst (m s)
```

We will use Haskell style syntactic sugar, so that using monads may feel more intuitive than using the explicit syntax; the kind of sugarization we use is based on two (purely cosmetic) simplifications, one for the *unit* operator and the other for the *bind* operator:

```
return ≡ unit
bind m (λv. ...) ≡ do v ← m
                            ...
```

3 Heap

We start by giving a definition of our notion of heap. The heap will be the basis for all our computations, in that it will provide the main, shared storage to which all statements and references in our programs refer to. We do not define our heap as a concrete type, but rather we give an “interface” (a Haskell type class) that specifies the operations that must be available on some generic type for it to be used in whichever context a heap is expected. Such an interface may also contain functions from types to types that specify some type transformations that our heap will need to offer. We define this predicate on a type variable h as:

```
Heap h
```

We start with type-level functions. A heap h must support a type function that, given h and a type α (some new item), returns a new heap; we will expect the returned heap to contain not only all the elements that the original heap h contained, but also a new slot for storing a value of type α:

```
New h : ∗→∗
```

Thanks to this definition we can now proceed to giving the two fundamental operators on a heap: *new* and *delete*, which respectively allow us to add an element to our heap and remove the last added element to our heap. The allocation and deallocation operators have the following types:

```
new : h→ α →New h α
del : New h α→h
```

4 References

Our notion of heap is too poor for our purposes. We immediately extend our heap predicate so that instead of having "just a heap" (which would be relatively pointless), whenever we have a heap h we also get a type ref for expressing references to within our heap h. This way we strongly couple the idea of some state (the heap itself) and the idea of doing mutable computations on it (the references we will define shortly). References are represented as pointing to some value α inside some heap h; because of this a reference will be represented with a functor ref that takes in two type parameters.

Our heap predicate now takes two type variables as parameters: the heap h and the functor ref; the signature now becomes:

```
Heap h ref
```

Followed by the various functions and operators that both h and ref must support. We give a small "calculus of references" that allows us to manipulate references by creating references from other references (selection), invoking methods from references (method invocation), and generating stateful statements from references. Statements that evaluate to a value of type α in a context with a heap h will have the form (known from the state monad):

```
ST h α
```

Some of the operators on references we will not introduce right away; these operators will be presented after some more background is covered. Two simple and basic operators on references are the evaluation and assignment operators:

```
eval : ref h α →ST h α
```

Which evaluates a reference to a statement that will return the same value contained inside the reference when passed a concrete heap, and:

```
(:=) : ref h α → α →ST h Unit
```

Which assigns a reference of type α a new value of type α and returns a statement that when evaluated will return a value of type $Unit$ (()) and the new heap where the assignment has been made.

Another kind of assignment operator is the in-place modification operator:

```
(*=) : ref h α → (α → α) →ST h Unit
```

which takes a reference of type α, a function of type $(\alpha \to \alpha)$ and returns a statement which, when passed a concrete value of the heap h, will evaluate the initial value of the reference, apply the function and return the result of assigning it to the original reference. This operator is quite handy for implementing common operators such as the in-place increment by one of an integer (++) or the in-place increment by some integer value (+ =) that are widely used in stateful programming languages such as C, Java or C#. The alternative to having such an operator is to evaluating and then assigning a reference; indeed, in light of this consideration we may realize that we already could define the body of this operator in terms of the state monad:

```
r *= f =
        do v ← eval r
           r := (f v)
```

It is interesting to realize that this definition is indeed extremely abstract, and implies absolutely nothing in terms of what is happening exactly inside the various operators: whether this operation simply writes a mutable state, synchronizes access to some shared memory location or even sends and receives some network messages with some other instance of the same distributed process is indeed unknown by just looking at this piece of code.

A very important operation on some reference ref and some heap h is the "introduction-and-elimination" operator that:

1. Allocates an initial value of type α on our heap h
2. Builds a reference to this value
3. Performs some arbitrary computation on this newly create reference, returning a result of type β
4. Cleans up the heap deleting the new value of type α before returning the β result

This operator is called $(\gg +)$, in order to recall the (similar) concept of binding in a monad. In fact, this operator will bind a new value of type α to a corresponding reference. The type of this operator is:

```
(>>+) : α →(ref (New h α) α →ST (New h α) β)→ST h β
```

We use the same representation that the state monad uses for values where our heap h is the mutable state of the computation. It is also interesting to notice that this operator works in a way that is very similar to a "scoping" operator. To realize this, notice that the value of type α is accessed through a reference that is exclusively available within the confines of the function that is passed as the second parameter to $(\gg +)$. As an example, let us consider the following pseudo-code:

```
A a = new A(...);
{
  B b = new B(...);
  {
    // do something with a and b
  }
}
```

Would be written, with our (>> +) operator:

```
A(...) >> + (λa.
B(...) >> + (λb.
  -- do something with a and b
))
```

5 First Examples:

Thanks to the system we have put together, we may already start writing some simple examples. These examples will only use the common operators that are required to exist for every heap h with references ref.

For the first example, let us say that we have a reference i to an integer within some heap h such that:

```
i : ref h Int
```

We can write:

```
ex₁ =
        do v ← eval i
           i := v+2
           v' ← eval i
           return v
```

The type of the code above is:

```
Heap h ref ⇒ ex₁ : ST h Int
```

The example above makes a quite limited use of our operators, but most importantly it assumes that a reference has already been created outside the example and is given. Let us show now how we can generate our local references in a self-contained example. The following example is more interesting since it shows the most reasonable pattern of introduction for references: we expect that a reference will be allocated with the $\gg +$ operator, used in its scope and then simply discarded, exactly as we would do in a "typical" imperative language. The example is:

```
ex₂ =
         do 10 >>= (λi.
         do "hello␣" >>= (λs.
         do s *= (λx.x++"world")
            v←eval s
            x←eval i
            return v ++ show x))
```

In this example we start by allocating an integer on the heap, then we allocate a string on the heap, and finally we manipulate the two references to these values on the heap before returning the concatenation of their values. Sadly, the example above does not compile without some modifications. We will see very shortly how we can write the example above so that at least it compiles correctly.

The last example we see is another term which does not type correctly. This time however this behavior is quite desired:

```
ex_wrong =
         do i' ← 10 >>= (λi.return i)
            i':=20
```

In this example we try to access a location on the heap that is not available, but luckily the last line cannot be typed since $i^{'}$:ref (New h_0 Int) Int and $i^{'}$?20 :ST (New h_0 Int) () but the expected value for the first type parameter is $h_0 \neq$ (New h_0 Int).