1 Methods

The same way we defined a selection operator for fields that allowed us to create a reference to a field from inside the record represented by a reference, we can give a selection oeprator for methods that allows us to select a method from a record that supports it (such a record is for all practical purposes an *Object* now) amd to invoke it. The two main challenges we will face are:

- making methods invocable through a simple application: no special invocation operation must be required
- making method selection and invocation possible thanks to an overload of the $\Leftarrow$ operator for field selection
- dealing with recursive definition for types

We start by giving the definition of a method that takes as input a value of type α and returns in output a value of type β and where "this" (the instance over which the method is invoked) has type τ:

```
Heap h ref ⇒ Method α β τ = Method (τ → α → (τ × β))
```

According to the above definition a method contains a function that takes as input a reference to a τ in some heap h, an α (the method parameter) and which retursn as output a reference to a value β in the same heap h. At this point we can give a definition for a first version of the selection operator; the selection operator will take a label that represents a method, a reference to a value that supports the label and returns an appropriate function that takes as input the method input and internally invokes the method on the input reference:

```
Heap h ref ⇒ (<=|) : ref h τ → Label τ (Method α β τ) → α ST h β
self_ref <=| (Label read write) =
    λx.do self ← eval self_ref
          let (Method m) = read self
                   (self',y) = m self x
          self_ref := self'
          return y
```

the method selection simply extracts the actual method m from $self$ (the evaluation of the input reference), applies the method to $self$, stores back the modified version of $self$ ($self'$) and returns the result of the method.

It is interesting to note is that this definition of methods is completely generic, since it does not require a complete implementation to be given thanks to the fact the concrete implementation can be nicely abstracted away with the set of operators given in the previous parts of this work.

At this point it would be interesting to give a way of building a method through our system of references rather than directly giving a definition that has the right signature. We would like to avoid making it necessary to manually care about the handling of the current value of *this* when writing methods that mutate the instance on which they are invoked. For this reason we define the operator:

```
Heap h ref ⇒ mk_method : (ref t t -> a -> ST t b) -> Method a t b
mk_method st_m =
    Method(λself.λx.
        let res = st_m id_ref x
        in res self)
```

where we require the presence of the identity reference (or reference to self):

```
Heap h ref ⇒ id_ref : ref a a
```

which in the simple mutable implementation would be

```
id_ref = Reference (λh.(h,h)) (λh'.λh.((),h'))
```

There is still a problem. Let us declare the type of a simple counter object:

```
Counter = Int : Method Unit Unit Counter
```

this declaration is recursive, and as such it does not work. We wish to fix this with the minimal overhead on the rest of the system, so we split the definition of *Counter* in two:

```
Counter k = Int : Method Unit Unit k
RecCounter = RecCounter (Counter RecCounter)
```

this way we both have a definition that still works with the rest of the system (*CounterRecCounter*) and a definition that we can use inside methods. We also wish to express the fact that the two definitions are equivalent, so we introduce an appropriate type predicate:

```
Recursive s
    Rec s : *
    cons : s -> Rec s
    elim : Rec s -> s
```

that captures the essential identity between a type (like our *CounterRecCounter*) and its recursive wrapper (*RecCounter*). We instance this predicate for our counter:

```
Recursive (Counter RecCounter)
    Rec (Counter RecCounter) = RecCounter
    cons = RecCounter
    elim (RecCounter x) = c
```

now we modify the signatures of method selection and method construction:

```
Heap h ref ∧ Recursive τ ∧ τ_rec = Rec τ ⇒ (<=|) : ref h τ → Label τ (Method α β τ_rec) → α ST h β
Heap h ref ∧ Recursive t ∧ t_rec = Rec t ⇒ mk_method : (ref t t -> a -> ST t b) -> Method a t_
```

and add the appropriate packing and unpacking to the definitions.

2 Selection Overload

Let us compare the signatures of the two selection operators we have seen until now:

```
Heap h ref ⇒ (<=) : ref h t -> Label t a -> ref h a
Heap h ref ⇒ (<=|) : ref h t -> Label t (Method a b t) -> a -> ST h b
```

it is clear that these two operators do similar things, and in fact they are quite similar. To be factored into one single operator we must:

- disambiguate the second type parameter of the label
- unify the return type ($refha$ rather than $(a- > STthb)$) with a type function

We give a simple type constructor:

```
Field a = Field a
```

that we will use to encapsulate all labels that select the fields of a record; we will now write:

```
Label r (Field a)
```

whenever we would have used a

```
Label r a
```

Now we give a simple type function that denotes the result of a selection:

```
SelectionResult : * → * → * → *
```

and we instance this function with the two only cases of selectables we have until now:

```
Heap h ref ⇒ SelectionResult h ref (Field a) = ref h a
Heap h ref ⇒ SelectionResult h ref (Method a b t) = a → ST h b
```

now we can express selection with a single operator:

```
Heap h ref ⇒ (<=) ref h t → Label t a → SelectionResult t a
```

The ⇒ operator (and the *SelectionResult* type function) are part of the *Selection* predicate, which is instanced twice.

We can easily fix the implementation of the selection operator to compensate for the additional need of packing and unpacking introduced by the *Field* constructor needed for the disambiguation above; also, we will need to apply a similar correction to the *labelAt* function.