Compositional References for Stateful Functional Programming

Koji Kagawa
Department of Information Science
Kagawa University
2-1 Saiwai-cho Takamatsu 760, Japan
E-mail: kagawa@ec.kagawa-u.ac.jp

Abstract

We introduce the notion of *compositional references* into the framework of monadic functional programming and propose a set of new primitives based on this notion. They enable us to use a wide range of mutable data structures. There, references may be passed around explicitly, or mutable data structures such as arrays and tuples may be passed implicitly as hidden state. The former style is called *the explicit style* and is usually more expressive, while the latter is called *the implicit style* and has simpler semantics. We investigate the relation between the two styles and discus implementation issues.

1 Introduction

Many proposals have been made toward a safe integration of a purely, lazy functional language with in-place updatable state [3, 1, *etc.*]. In a series of proposals [16, 11], the notion of *monads* [7] provides a basis for such an integration.

Based on monads, Launchbury and Peyton Jones [5] proposed a way to express computations which deal with multiple mutable objects by providing primitives for ML-like references, and at the same time to securely encapsulate such computations by using rank-2 polymorphism. Such references are passed explicitly in stateful[1] programs.

On the other hand, in Wadler's former proposal [16], data structures such as arrays are used directly as state and passed implicitly. No explicit reference is passed around. We will refer to, in what follows, Wadler's style, *implicit* style and Launchbury and Peyton Jones's style, *explicit* style. Each style has its merits and demerits:

- The implicit style has a simpler functional account. We do not need the notion of "global states" in order to explain the behaviour of programs.
- The explicit style enables us to write stateful programs in a traditional imperative fashion. In addition, it is, in general, more expressive.

[1] Following [5], we use the term *stateful* to refer to computations in which we would like to deal with state destructively.

To appear in the International Conference on Functional Programming 1997, June 1997, Amsterdam, the Netherlands.

Having simpler types is sometimes advantageous in expressiveness. As we will see later, Launchbury and Peyton Jones's approach requires all state transformers to be polymorphic with respect to the state parameter in order to confine references. This prevents state transformers from being used as function parameters in languages based on the Damas-Milner type system. In general, however, the implicit style is less expressive. First of all, it is impossible to deal with multiple mutable objects in Wadler's original proposal. And even if it is possible, it seems at least difficult to express mutable *graph* structures without introducing a mechanism equivalent to references. Therefore, it would be desirable to enjoy merits of both styles in a single program. So far, however, there has been no proposal which makes such mixed style possible.

In this paper, we will propose a set of new primitives into the monadic style of functional programming in order to extend both styles and to make it possible for us to enjoy merits of the two styles. As a result, we will be able to write state transformers more concisely and to use a wide range of mutable data structures. We introduce the notion of *compositional references*[2] as a key mechanism.

The idea of compositional references is as follows. Instead of introducing mutable data types as new data types independent of existing immutable ones, we introduce a "mutability wrapper" `Mutable` as a type constructor so that when `Array a` is the type of ordinary (immutable) arrays, `Mutable s (Array a)` is the type of mutable arrays with state parameter `s`. We will call such mutable data structures compositional references. We also provide some associated operators to combine such compositional references with state transformers.

Instead of introducing compositional references as an ad-hoc extension, we will give functional account of the type constructor and associated operators. Compositional References are essentially *expansion morphisms* in the category of *state shapes* [13, 9], which were introduced in order to explain block structures of Algol-like languages. Pierce and Turner [12] and Hofmann and Pierce [2] also used the same mechanism in order to explain inheritance and subtyping in object-oriented programs. Using functional account, we can reason about the behaviour of stateful programs as ordinary functions.

The rest of the paper is organized as follows. In Section 2, we explain monadic functional programming and its two styles more in depth. Then, in Section 3, we introduce

[2] In the earlier version which appeared in SIPL'95 [4], they were called *composable references* and `Mutable` below was written as `CR`.

some new primitives and the notion of *compositional references*. In Section 4, we formulate the relation between the two styles. Then, Section 5 discusses implementation issues. Section 6 concludes. We will use the syntax of Haskell [10] in the following.

2 Previous Work

In this section, we explain previous work on monadic functional programming and describe the difference between Wadler's style and Launchbury and Peyton Jones's style.

2.1 Implicit style

It has been shown that monads provide a uniform framework for various forms of sequential computations (i.e. those with *side-effects*). And they have been popularized in the functional programming community. For motivations and examples, see [7, 15, 16].

When we would like to deal with "state," we use the following data type:

```
type ST s a = s -> (a, s)
```

with monadic constructs such as:

```
returnST :: a -> ST s a
returnST a = \ s -> (a, s)

thenST :: ST s a -> (a -> ST s b) -> ST s b
m `thenST` k = \ s0 -> let (a, s1) = m s0
                       in k a s1
```

Values of type `ST s a` (called *state transformers*) represent computations which modify state of type `s` and produce results of type `a`. The operator `returnST` does not change state and simply returns a value which is supplied as the argument. The operator `thenST`[3] is used to sequentialize two state transformers. The monad of state transformers is considered to be a suitable abstraction in order to hide state from programmers, and to sequentialize accesses to state. We hide the implementation of state transformers from programmers and carefully provide read/write primitives so that they do not destroy single-threadedness of state. Then, it becomes possible to design primitives so that they update data structures in place.

In order to deal with mutable arrays, Wadler [15] proposed using the following array primitives[4]:

```
readArr  :: Int -> ST (Array a) a
writeArr :: Int -> a -> ST (Array a) ()
```

which read and overwrite an array in the index given as the first argument, and:

```
initST   :: s -> ST s x -> x
initST = \ s st -> fst (st s)
```

which gives an initial state to state transformers. Note that the type parameter `s` is instantiated to specific types such as `Array Int` here. This contrasts with Launchbury and Peyton Jones's proposal explained next.

[3] We will use `thenST` as an infix operator in the following using the Haskell notation `` `_` `` for infix operators. In Haskell 1.3, `thenST` is written as `>>=`.

[4] The names and the types of primitives differ in an inessential way from the original ones.

2.2 Explicit style

Launchbury and Peyton Jones's proposal also uses the monadic framework explained above. However, in their proposal, the state type parameter of `ST` is used for a rather technical reason. They proposed a "primitive" operator named `newVar` which creates a new ML-like *reference* (of type `MutVar s a`):

```
newVar   :: a -> ST s (MutVar s a)
```

and associated operators `readVar` and `writeVar` which respectively read and update a reference.

```
readVar  :: MutVar s a -> ST s a
writeVar :: MutVar s a -> a -> ST s ()
```

Here, the type of references (`MutVar s a`) is parameterized over the type of the state (`s`) as well as the type of the object being referred to (`a`). Though `s` is intuitively the type of the global state, it is used in a special typing rule explained next which prevents references from escaping their scopes.

In order to execute state transformers, we use a *built-in construct* (it is *not* an ordinary function) named `runST` of the following special type:

```
runST :: ∀a. (∀s. ST s a) -> a
```

When `runST` is applied to a state transformer, it passes an "empty" initial state to the state transformer and then extracts its final result. In this way, `runST` extracts a pure value from a stateful computation. The special typing above is necessary for the following reason. If a reference created in one state transformer could be used in another transformer, the result of the program would depend on the evaluation order. Detecting errors caused by such *cross* pointers at runtime would be expensive. By requiring state transformers to be polymorphic with respect to the state parameter, they proved, using parametricity, that values within one state transformer cannot depend on references generated by other state transformers and therefore that it is safe to interleave evaluation of such state transformers.

They also gave primitive operators for mutable arrays (`MutArr s a`). A simplified version of the array operators is shown below, where arrays are indexed by integers.

```
newArr   :: Int -> a -> ST s (MutArr s a)
readArr  :: MutArr s a -> Int -> ST s a
writeArr :: MutArr s a -> Int -> a -> ST s ()
```

The first argument of `newArr` is the size of the newly created array. This operator creates a new mutable array. The other two operators are used to read and overwrite the array given as the first argument in the index given as the second argument. The type of mutable arrays (`MutArr s a`) is also parameterized over the type of the state (`s`), and its second parameter (`a`) is the type of array elements. In their proposal, programs which deal with mutable arrays have type `MutArr s a -> ... -> ST s x`.

Mutable arrays (`MutArr s a`) are different from arrays of references (`Array (MutVar s a)`)—they have different representations [5, Section 3]. `MutArr s a` is more space-efficient since it is directly mutable and does not require indirection cells. On the other hand, `Array (MutVar s a)` needs indirection cells, however, it is more expressive since it can represent shared mutable cells. (For example, the second and the third elements can be the same reference. In such a case, if we modify the second element, the third element is modified simultaneously due to sharing.)

3 Extensions

In the implicit style, we were not able to deal with more than one piece of state. This is why Launchbury and Peyton Jones introduced new primitives and an encapsulation mechanism for them. However, when we deal with only one mutable array, the implicit style seems to have an advantage—programs have simpler accounts and types.

In this section, we will introduce some new primitives so that we can write stateful programs enjoying advantages of both styles.

3.1 Compositional References

For the moment, we introduce the following *primitive* data type:

```
type  Mutable s t
```

as a simple generalization of `MutArr` to general data structures other than arrays. It stands for the mutable version of type `t` with a state parameter `s`. Then, `MutArr s a` is a type synonym of `Mutable s (Array a)`. For example, `Mutable s (a, b, c)` is the type of mutable triples. We will call values of type `Mutable s t` *compositional references from* `s` *to* `t`. The reason for this name will be explained later.

`Mutable s (a, b, c)` and `(MutVar s a, MutVar s b, MutVar s c)` are different in the same way as `MutArr s a` and `Array (MutVar s a)` are different. That is, the former is the type of directly mutable triples, while the latter is the type of triples of references. Needless to say, `Mutable s (a, b, c)` and `MutVar s (a, b, c)` (more generally, `Mutable s t` and `MutVar s t`) are different types. The former is the type of mutable triples, while the latter is the type of references to immutable triples. Actually, `MutVar s a` is a special case of `MutArr s a` where the size of the array is exactly one. Therefore, we define the following data type as a record type with only one field.

```
data Atom a = MkAtom a
```

Then, `MutVar` can be explained as follows.

```
type MutVar s a = Mutable s (Atom a)
```

Our intention here is that `Mutable s _` makes only *toplevel* fields of data structures mutable. For example, `Mutable s (a, (b, c))` is the type of *mutable* pairs whose second components are *immutable* pairs, while `Mutable s (a, b, c)` is the type of *mutable* triples. Therefore, they are not equivalent. The reason for this design decision will be discussed later in Section 5.

Note that there is a correspondence between implicit style state transformers and explicit style ones as follows:

Implicit style:	`... -> ST t x`
Explicit style:	`... -> Mutable s t -> ST s x`

For example, the array updating operation has type `Int -> a -> (Array a) ()` in the former, while it has type `Int -> a -> Mutable s (Array a) -> ST s ()` in the latter (ignoring the order of parameters). This correspondence will play an important role in the following development.

3.2 Composing References and State Transformers

Next, we introduce some operators associated with compositional references.

We would like to combine an array reference `ar` (of type `Mutable s (Array a)`) with a state transformer of the implicit style `st` (of type `ST (Array a) x`) so that they can produce a state transformer of type `ST s x`. For this purpose, we introduce the following operator:

```
appR :: Mutable s t -> ST t x -> ST s x
```

When `t` is the type of arrays, it takes an array reference `ar` and a state transformer for arrays `st` and produces a state transformer for *the global state* (`s`) in which the referenced array resides. That is, `ar` ‘`appR`‘ `st`[5] has type `ST s x` and is a global state transformer which only affects the referenced array and has no effect on the rest of the state. Though `appR` deals with references, we will still call this style *implicit*, if state transformers use only `appR` and the operators introduced in the next subsection, since mutable data structures are finally passed to state transformers as their hidden parameters.

From the implementation point of view, `Mutable s t` should be a primitive data type since we would like its values to be represented as efficiently as possible. Intuitively, they should be direct pointers to mutable data structures. Still, we can consider a functional account of compositional references, like that of state transformers. We introduce two auxiliary functions for this purpose:

$$\begin{aligned} rd &: \mathit{Mutable}\, s\, t \to s \to t \\ wr &: \mathit{Mutable}\, s\, t \to s \to t \to s \end{aligned}$$

which read and overwrite the substructure of type t in the structure of type s. (In other words, we can view `Mutable` as the following data type:

$$\textbf{type}\ \mathit{Mutable}\, s\, t \quad = \quad (s \to t, s \to t \to s)$$

when we explain their behaviour.)

We require that these two functions satisfy some natural equations taken from [2],

$$rd\, r\, (wr\, r\, s\, t) = t \tag{1}$$

$$wr\, r\, s\, (rd\, r\, s) = s \tag{2}$$

$$wr\, r\, (wr\, r\, s\, t_1)\, t_2 = wr\, r\, s\, t_2. \tag{3}$$

Then $appR$ can be defined using these two functions.

$$r\ \text{`}appR\text{`}\ st \stackrel{\text{def}}{=} \lambda s \to \ \textbf{let}\ t = rd\, r\, s;\ (a, t') = st\, t\ \textbf{in}\ (a, wr\, r\, s\, t') \tag{4}$$

Intuitively, `appR` extracts the data structure of type `t` from the state (of type `s`), and applies the state transformer for type `t` to the extracted substructure. In other words, compositional references of type `Mutable s t` can “extend” the state parameter of state transformers from a smaller one (`t`) to a larger one (`s`).

Global states “conjured up” by `runST` can be considered as very large data structures such as:

$$\begin{aligned} s_0 \stackrel{\text{def}}{=} (&a_{1,1}, \ldots, a_{1,m_1},\ \ldots, a_{n-1,m_{n-1}}, \\ &a_{n,1}, \ldots, a_{n,m_n}, \\ &a_{n+1,1}, \ldots,\ a_{N,1}, \ldots, a_{N,m_N}) \end{aligned}$$

[5] We assume ‘`appR`‘ binds tighter than ‘`thenST`‘.

If a state transformer `st :: ST (Array a) ()` transforms $(a_{n,1},\ldots,a_{n,m_n})$ to $(a'_{n,1},\ldots,a'_{n,m_n})$ and `cr :: Mutable s (Array a)` is a compositional reference which points to $a_{n,1},\ldots,a_{n,m_n}$ in s_0:

$$\begin{aligned} rd\ cr\ s_0 &= (a_{n,1},\ldots,a_{n,m_n}) \\ wr\ cr\ s_0\ (a'_{n,1},\ldots,a'_{n,m_n}) &= \\ &(a_{1,1},\ldots,a_{1,m_1},\ \ldots,a_{n-1,m_{n-1}}, \\ &\boxed{a'_{n,1},\ldots,a'_{n,m_n}}, \\ &a_{n+1,1},\ldots,\ a_{N,1},\ldots,a_{N,m_N}) \end{aligned}$$

their composition `cr 'appR' st` transforms s_0 to

$$(a_{1,1},\ldots,a_{n-1,m_{n-1}},\boxed{a'_{n,1},\ldots,a'_{n,m_n}},a_{n+1,1},\ldots,a_{N,m_N}).$$

As mentioned in the introduction, compositional references are known in semantics of imperative programs as *expansion morphisms* in the category of *state shapes* [13, 9]. They are used in order to explain block structures of imperative languages. What we do in this paper is to use them as first-class objects in order to deal with various data types as state in monadic functional programs—rather than as a concept in semantics.

Pierce and Turner [12] used pairs of functions for extracting ($get : s \to t$) and overwriting ($put : s \to t \to s$) substructures in order to explain *inheritance* in object-oriented programs. Hofmann and Pierce [2] directly associated such pairs to subtyping. Our use of compositional references is similar to theirs of such pairs of functions. However, their main interest is to give a type-theoretic foundation of object-oriented programming. In contrast, we use compositional references as a generalization of references and hence, as first-class values.

An advantage of the implicit style state transformers is that they have simple monomorphic types. As we mentioned in the introduction, Launchbury and Peyton Jones's method requires state transformers to be polymorphic with respect to the state, which sometimes prevents state transformers from being passed as function parameters. For example, it is impossible to type-check the following function.

```
{-
useArr ::
   (∀ s. MutArr s Int -> ST s ()) -> Int
useArr ast = runST
               (newArr 10 0 `thenST` \ ar ->
                ast ar      `thenST` \ _  ->
                readArr ar 0)
-}
```

This is because `runST` requires the state parameter to be polymorphic while `ast` is used as a function parameter and must be monomorphic. On the other hand, if we define a state transformer (`ast'`) of type `ST (Array Int) ()`, it is possible to pass them as a parameter of a function such as above and then to use it as `ar 'appR' ast'` in the function.

Another possible solution to this problem would be to extend the type checker as described in [8] and to use rank-2 types in type signatures. We will return to this point later.

Once we introduce `appR`, we have to slightly modify the type of primitives such as `newVar` and `newArr` which create new mutable data structures . They can no longer be given unrestricted polymorphic types, since specific data types such as arrays are used as the state parameter of state transformers. Here, we introduce a *built-in (primitive)* type class `Global s` in order to show that `s` is the type of global states "conjured up" by `runST` and that it has an ability to return any number of and any type of fresh locations for mutable data structures[6].

```
newVar :: Global s => a -> ST s (MutVar s a)
newArr ::
   Global s => Int -> a -> ST s (MutArr s a)
```

When global state transformers are applied to `runST`, the type constraint `Global s` is eliminated. Therefore, the type of `runST` should be:

```
runST :: ∀a. (∀s. Global s => ST s a) -> a
```

In general, such type constraints are considered as hidden parameters. In the case of `Global s`, it would be a function which returns a fresh location when applied to the global state.

Instead of introducing new primitives for each data type such as `newPair` and `newTriple`, we introduce the following primitive which creates a mutable version of its argument by creating the copy of its *toplevel* fields.

```
newMutable :: Global s => a -> ST s (Mutable s a)
```

For example, `newMutable (1,2)` creates a new mutable pair in the global state. Of course, it would be possible to avoid copying when the argument of `newMutable` is a constructor application like `(1,2)`. Since its behaviour depends on the size and therefore on the type of its argument, it might be better to constrain its type with say, `Copyable a` as well[7].

3.3 Intermediate References

In the implicit style, we will need primitives corresponding to `readVar` and `writeVar` in order to define state transformers for each data type. In the design of such primitives, we can use the correspondence of types between implicit and explicit styles explained in Section 3.1.

Implicit style: `... -> ST t x`
Explicit style: `... -> Mutable s t -> ST s x`

Then, the primitive read/write operators in the explicit style

```
readVar  :: MutVar s a -> ST s a
writeVar :: MutVar s a -> a -> ST s ()
```

correspond to the following new primitives in the implicit style respectively:

```
fetch  :: ST (Atom a) a
assign :: a -> ST (Atom a) ()
```

Their behaviour can be explained as follows:

```
fetch = \ (MkAtom a) -> (a, MkAtom a)
assign a' = \ (MkAtom a) -> ((), MkAtom a')
```

(For the moment, we ignore the issue of in-place updating.)

For example, the following state transformer increments `Atom` cells.

[6] Another possible way would be to include the constraint in the type of the state as follows.
`newVar :: a -> ST (Global s) (MutVar (Global s) a)`

[7] If we could use multi-parameter type classes, type constraint of the form `Global2 s a` might be better.

```
incrST :: ST (Atom Int) ()
incrST = fetch       `thenST` \ n ->
         assign (n+1)
```

In order to apply them to, say, the first field of triples, we need a function of type:

```
ST (Atom a) x -> ST (a, b, c) x
```

However, we notice that this is exactly what a compositional reference of type `Mutable (a, b, c) (Atom a)` and `appR` can generate. Therefore, we provide the following compositional references as primitives for triples:

```
fst3R :: Mutable (a, b, c) (Atom a)
snd3R :: Mutable (a, b, c) (Atom b)
thd3R :: Mutable (a, b, c) (Atom c)
```

In practice, such compositional references should be provided automatically, when new data types are defined. One possible way is to extend the Haskell declaration of labeled fields [10, Section 4.2.1] so that the following declaration:

```
data C = F {f1,f2 :: Int, f3 :: Bool}
```

generates three compositional references `f1`, `f2` and `f3` of type `Mutable C (Atom Int)`, `Mutable C (Atom Int)` and `Mutable C (Atom Bool)` respectively.

Such compositional references stand for *relative* locations of fields *with respect to* the base structure. For example, the functional account of `fst3R` is given as follows:

$$\begin{aligned} rd\, fst3R &= \lambda(a,b,c) \to (MkAtom\, a) \\ wr\, fst3R &= \lambda(a,b,c) \to \lambda(MkAtom\, a') \to (a',b,c) \end{aligned}$$

The expression `fst3R `appR` incrST` has type `ST (Int, b, c) ()` and it changes the state from (n,b,c) to $(n+1,b,c)$.

In practice, `fst3R` would have a representation quite different from those references which are created by operators such as `newArr` and `newMutable`. In order to distinguish the two kinds of references explicitly, we use the term *global* references for those which are created by `newMutable` and so on and *intermediate* references for `fst3R`, `snd3R` and so on.

We can also think of compositional references which stand for the location of substructures more general than fields. For example, we can think of a compositional reference `fstSnd3R` which stands for the location of the first and the second fields of triples. It has type `Mutable (a, b, c) (a, b)` and its behaviour can be explained by the following equations.

$$\begin{aligned} rd\, fstSnd3R &= \lambda(a,b,c) \to (a,b) \\ wr\, fstSnd3R &= \lambda(a,b,c) \to \lambda(a',b') \to (a',b',c) \end{aligned}$$

When the Haskell data type declaration is extended to support some form of inheritance, it is probable that such compositional references are provided when new data types are defined by *inheriting* existing ones. Then, compositional references such as `Mutable (a,b,c) (a,c)` where the mutable piece is not contiguous cannot be introduced. In this paper, we do not discuss particular syntax for inheritance, though.

For example, let

```
fstR :: Mutable (a, b) (Atom a)
sndR :: Mutable (a, b) (Atom b)
```

be two compositional references for the two fields of pairs and let `rotST t` be a state transformer for pairs which transforms (x,y) to $(x\cos t - y\sin t, x\sin t + y\cos t)$:

```
rotST  :: Double -> ST (Double, Double) ()
rotST t = fstR `appR` fetch     `thenST` \ x ->
          sndR `appR` fetch     `thenST` \ y ->
          fstR `appR` assign (x*cos t - y*sin t)
                                `thenST` \ _ ->
          sndR `appR` assign (x*sin t + y*cos t)
```

we can reuse `rotST` in the definition of `spiralST`, a state transformer for triples which transforms (x,y,z) to $(x\cos t - y\sin t, x\sin t + y\cos t, z+t)$.

```
spiralST  ::
       Double -> ST (Double, Double, Double) ()
spiralST t = fstSnd3R `appR` rotST
                                `thenST` \ _ ->
             thd3R  (fetch `thenST` \ z ->
                     assign (z+t))
```

Note that though they are written imperatively, their functional accounts can be given as follows:

$$\begin{aligned} rotST\ t &= \lambda\ (x,y) \to \\ &\quad ((), (x\cos t - y\sin t, x\sin t + y\cos t)) \\ spiralST\ t &= \lambda\ (x,y,z) \to \\ &\quad ((), (x\cos t - y\sin t, x\sin t + y\cos t, z+t)) \end{aligned}$$

If they are written in the explicit style, such accounts would involve the global state, even though its most part is not relevant.

3.4 An Example

Here, we give a small example in the implicit style which uses primitives introduced so far. For arrays, we assume that `arrIdx` is a primitive such that the expression `arrIdx i` stands for the location of the $\mathtt{i}^{th}$ element.

```
arrIdx :: Int -> Mutable (Array a) (Atom a)
```

Then `applyST` applies the given state transformer according to the given list.

```
applyST ::
     [Int] -> ST (Atom a) () -> ST (Array a) ()
applyST []     st  = returnST ()
applyST (i:is) st  =
            arrIdx i `appR` st `thenST` \ _ ->
            applyST is st
```

If the supplied list is $[1,2,\ldots,n]$ where n is the number of elements, `applyST` behaves as the "apply to all" function. Suppose that `anArr :: MutArr s Int` is a mutable array of size three with all elements initialized to `0`, then the expression `anArr `appR` (applyST [1, 2, 2, 2, 2, 1, 3] incrST)` counts the frequency of the elements of the given list—the array is modified so that its first, second and third elements are 2, 4 and 1 respectively. Of course, an equivalent program can be written in the explicit style. The advantage of the implicit style here is that it is obvious from the type that `applyST` only affects the array and does not interfere with the rest of the state. Another advantage is that if we provide a variant of `initST` as follows,

```
initST' :: s -> ST s x -> (x,s)
initST' = \ s st -> (st s)
```

it becomes possible to use the modified state in the rest of the program.

3.5 Compositional References in the Explicit Style

In the explicit style, the following primitive operator:

```
cmpR ::
   Mutable s t -> Mutable t u -> Mutable s u
```

which composes two compositional references of appropriate types will be useful. The name *compositional reference* comes from this operator. Intuitively, it simply returns the sum of two relative locations. Its behaviour is characterized by the equations below:

$$rd\,(r\;`cmpR`\;q) = (rd\,q) \circ (rd\,r) \quad (5)$$

$$wr\,(r\;`cmpR`\;q)\,s\,u = wr\,r\,s\,(wr\,q\,(rd\,r\,s)\,u) \quad (6)$$

$$p\;`cmpR`\;(q\;`cmpR`\;r) = (p\;`cmpR`\;q)\;`cmpR`\;r \quad (7)$$

Then, `\ x -> x `cmpR` fst3R` gives the reference to the first field of the mutable triple `x`.

For example, if we write `rotST` in the explicit style, it becomes as follows:

```
rotST' :: Double ->
          Mutable s (Double, Double) -> ST s ()
rotST' t = \ r ->
   readVar  (r `cmpR` fstR)     `thenST` \ x ->
   readVar  (r `cmpR` sndR)     `thenST` \ y ->
   writeVar (r `cmpR` fstR)
            (x*cos t - y*sin t) `thenST` \ _ ->
   writeVar (r `cmpR` sndR)
            (x*sin t + y*cos t) `thenST` \ _ ->
```

In the (extended) explicit style, we use `cmpR` instead of `appR`. All operations are carried out with respect to references, instead of the state. Only `readVar` and `writeVar` operates upon the state.

With `cmpR`, compositional references such as `fstSnd3R` can behave as cast operators of references by hiding some part of mutable data structures. Suppose we have `pt :: Mutable s (Int, Int)` and `cpt :: Mutable s (Int, Int, Color)`, then `pts = [pt, cpt `cmpR` fstSnd3R]` has type `[Mutable s (Int, Int)]`. Borrowing from object-oriented terminology, `cmpR` is used for *subtyping*, while `appR` is used for *inheriting* methods.

In a sense, `appR` (or more precisely, `flip appR` where `flip f x y = f y x`) can be thought of as a transformation from the implicit style to the explicit style. In the other direction, it becomes also possible to convert (polymorphic) explicit style state transformers into implicit style ones, if we provide a *built-in* construct of type:

```
extendST ::
(∀ s. Global s => Mutable s t -> ST s a) -> ST t a
```

In the explicit style, we can freely use operators such as `newVar` and can extend the state temporarily. In this way, we can go between the two styles and enjoy merits of both styles.

We can use `extendST` also when the state of type `t` is created by `runST` or another `extendST`. In that case, `extendST` can be viewed as a generalization of `runST` [8] and as an operator which extends the global state temporarily. It creates a new global state which is empty except that it contains the existing state of type `t`. Then, it passes the new state to the given state transformer with a compositional reference designating the location of the existing state. After execution, it throws away the newly created part of the state and behaves a state transformer for the state of type `t`. Without the compositional reference of type `Mutable s t`, two states of type `s` and `t` would be irrelevant—it would be impossible to access the existing state (of type `t`) from the state transformer for the extended state (of type `s`).

4 Relation between the Two Styles

At first, programs which are written in the implicit style and which use `appR` may seem inefficient since the functional account of `appR` given in Section 3.2 conceptually requires copying substructures before passing them to state transformers. For example, the expression `fstSnd3R `appR` rotST t` requires, at least conceptually, copying of the first and the second fields of triples. On the other hand, if we write programs entirely in the explicit style, we would not need such copying. For example, if `trR :: Mutable s (a, b, c)`, the expression `trR `cmpR` fstSnd3R :: Mutable s (a, b)` can probably be implemented only by simple pointer arithmetic on references. States are not touched in the meantime and are accessed only via `readVar` and `writeVar`. They are single-threaded, and state transformers can be destructive in the explicit style.

Even in the implicit style, it seems possible to implement `appR` so that it takes as an argument a pointer to a triple, simply adds the offset corresponding to `fstSnd3R` and then passes the result pointer to state transformers, since state transformers should be designed so that they update states destructively. However, we have not discussed in-place updating in the implicit style, so far.

The purpose of this section is to show that there is a certain relation between the two styles and therefore that the implicit style programs can be implemented as the corresponding explicit style ones. The relation itself can be shown irrespective of in-place updating. Then it will be shown that in-place updating is also possible in the implicit style.

The key is the equations derived from the ones given in Section 3.

$$p\;`appR`\;(m\;`thenST`\;k) = (p\;`appR`\;m)\;`thenST`\;\lambda a \to p\;`appR`\;k\,a \quad (8)$$

$$p\;`appR`\;(returnST\,a) = returnST\,a \quad (9)$$

$$p\;`appR`\;(q\;`appR`\;m) = (p\;`cmpR`\;q)\;`appR`\;m \quad (10)$$

Equations (8) and (9) state that $(p\;`appR`)$ behaves as a *monad morphism*. For the exact definition and examples of monad morphisms, please refer to [15]. The equation (8) states that $`appR`$ can distribute over $`thenST`$ and (10) says that $`appR`$ can be substituted by $`cmpR`$ in some cases. We will make use of these facts in order to give efficient implementation of implicit style state transformers.

[8] And it is a generalization of the operator (of type `(∀ s. ST s a) -> ST t a`) which Peyton Jones proposed in his invited talk at SIPL'95.

For this purpose, in addition to the standard interpretation of state transformers:

$$\textbf{type}\ ST_{cp}\ s\ a \quad = \quad s \to (a, s)$$

we consider an alternative interpretation of state transformers:

$$\textbf{type}\ ST_{nc}\ s\ a \quad = \quad \mathit{Mutable}\ G\ s \to G \to (a, G)$$

where G is some fixed data type for global state and therefore $\mathit{Mutable}\ G\ s$ is the type of global references. (So far, the type of global state is abstract in order to make encapsulation of references possible. Therefore, we can think of the type above as the following rank-2 type.

$$\textbf{type}\ ST_{nc}\ s\ a \quad = \quad \forall g.\ \mathit{Mutable}\ g\ s \to g \to (a, g)$$

Here, however, it is considered as a fixed and universal data type in order to concentrate on the essential issue.) We distinguish the two semantics by subscripting their components by *cp* (for *copy*) or *nc* (for *no copy*). Then we give meanings of the primitives in these two semantics so that they are properly related and that we obtain the same results for values of observable types. The point is that primitives are defined in the non-copying semantics so that, roughly speaking, the following relation holds between the two semantics of state transformers:

$$st_{nc} \quad = \quad \lambda p \to\ p\ `\mathit{appR}_{cp}`\ st_{cp} \qquad (11)$$

Here st_{nc} and st_{cp} are the meanings of some term of type $ST\ s\ a$ in the copying and the non-copying semantics respectively.

The meanings of primitive state transformers including *fetch* and *assign* are given so that they satisfy the relation.

$$\mathit{fetch}_{nc} \quad \stackrel{\text{def}}{=} \quad \lambda p \to p\ `\mathit{appR}_{cp}`\ \mathit{fetch}_{cp} \qquad (12)$$

$$\mathit{assign}_{nc}\ a \quad \stackrel{\text{def}}{=} \quad \lambda p \to p\ `\mathit{appR}_{cp}`\ \mathit{assign}_{cp}\ a \qquad (13)$$

Note that in the non-copying semantics they are almost the same as *newVar* and *writeVar* except the order of arguments.

The other primitives are defined as:

$$q\ `\mathit{appR}_{nc}`\ m' \quad \stackrel{\text{def}}{=} \quad \lambda p \to m'\ (p\ `\mathit{cmpR}`\ q) \qquad (14)$$

$$m'\ `\mathit{thenST}_{nc}`\ k' \quad \stackrel{\text{def}}{=} \quad \lambda p \to m'\ p\ `\mathit{thenST}_{cp}`\ \lambda a \to k'\ a\ p \qquad (15)$$

$$\mathit{returnST}_{nc}\ a \quad \stackrel{\text{def}}{=} \quad \lambda_ \to \mathit{returnST}_{cp}\ a \qquad (16)$$

Note that we use the convention in the definitions above and the equations below that p and q are compositional references of appropriate types and:

$$\begin{array}{lcl} m & : & s \to (a, s) \\ m' & : & \mathit{Mutable}\ G\ s \to G\ \to (a, G) \\ k & : & a \to s \to (b, s) \\ k' & : & a \to \mathit{Mutable}\ G\ s \to G\ \to (b, G). \end{array}$$

The definition of appR_{nc} (14) is the key step to avoid copying since appR_{cp} requires copying of substructures while cmpR, and hence appR_{nc} require only simple pointer calculation. These definitions are justified by the following equations (8), (9) and (10). Then from these equations, we can show that if we define primitives as in (14), (15) and (16), the following equations hold between the meanings of appR, thenST and $\mathit{returnST}$ in the two semantics, where we write m^* for $\lambda p \to p\ `\mathit{appR}_{cp}`\ m$.

$$\begin{array}{rcl} (q\ `\mathit{appR}_{cp}`\ m)^* & = & q\ `\mathit{appR}_{nc}`\ m^* \\ (m\ `\mathit{thenST}_{cp}`\ k)^* & = & m^*\ `\mathit{thenST}_{nc}`\ (\lambda a \to (k\ a)^*) \\ (\mathit{returnST}_{cp}\ a)^* & = & \mathit{returnST}_{nc}\ a \end{array}$$

From these equations and the design of the other constants, state transformers constructed from them satisfy the relation (11).

More precisely, we need to define a type-indexed family of *logical relations* [6, Chapter 8] between the copying and the non-copying semantics. The relations are defined by induction on types as follows:

$$\begin{array}{rcl} c \sim_b c & & (b : \text{base type}) \\ e \sim_{\tau_1 \to \tau_2} e' & \stackrel{\text{def}}{=} & \textbf{for any}\ e_1 \sim_{\tau_1} e_1', \\ & & \quad e\ e_1 \sim_{\tau_2} e'\ e_1' \\ (e_1, e_2) \sim_{(\tau_1, \tau_2)} (e_1', e_2') & \stackrel{\text{def}}{=} & e_1 \sim_{\tau_1} e_1'\ \textbf{and}\ e_2 \sim_{\tau_2} e_2' \end{array}$$

And the relation for the type constructor ST is defined as follows.

$$\begin{array}{rcl} st \sim_{ST\ \sigma\ \alpha} st' & \stackrel{\text{def}}{=} & \textbf{for any}\ s, g, q, \\ & & \textbf{if}\ s \sim_\sigma rd\ q\ g\ \textbf{then} \\ & & st\ s \sim_{(\alpha,\sigma)} (a, rd\ q\ g_1)\ \textbf{and} \\ & & wr\ q\ g_1\ (rd\ q\ g) = g \\ & & \quad \textbf{where} \quad (a, g_1) \quad = \quad st'\ q\ g \end{array}$$

Then we can show:

Lemma: All the constants (*returnST*, *thenST*, *appR*, *fetch*, and *assign*) satisfy the logical relations.

Proof: (See Appendix A.)

Then from the basic lemma of logical relations, terms built from such constants all satisfy the relations. Especially, for a closed program with an observable type such as *Int*, its meanings in the two representations are the same. This result shows that implicit style programs can be implemented efficiently as explicit style ones that do not need copying of substructures.

As for safety of in-place updating, the discussion in [5] can be applied to our set of primitives. That is, in-place updating is safe if primitives are designed so that the state is used only in a single-threaded manner and if all read/write primitives are *strict* in the state. Of course, our read/write primitives (`fetch` and `assign`) can be implemented so that they satisfy these requirements. Note that all read/write primitives operate on fields and that the contents of fields themselves are not mutable. Therefore, there is no operator that duplicates state.

5 Discussion

In this section, we give some remarks on implementation issues.

From the result of the previous section, we can draw the following implementation strategy.

First, state transformers should take a reference as an argument and then modify the state destructively. For uniformity, even global state transformers should take a dummy

parameter. Second, compositional references are divided into three kinds—"ordinary" global references created by `newMutable`, intermediate references such as `fstR` and `sndR`, and global references produced by composing references of the first and the second kinds. Ordinary global references are represented as before (i.e. as pointers). Interior pointers (i.e. pointers into the middle of data structures) may be necessary for references of the third kind, which may complicate garbage collectors. In that case, they can be represented as a pair of a pointer and an offset with a special tag. Note that this difficulty always arises when we would like to deal with pointers into the middle with garbage collection, and therefore is not specific to our framework.

Intermediate references are represented as a function from global references to global references also with a special tag in order to be distinguished from ordinary global references. Then, `cmpR` just applies this function to the global reference given as the first argument. If we do not need to deal with variant types, intermediate references may be represented simply as an offset.

In the rest of this section, we explain why `Mutable s _` makes only toplevel fields of data structures mutable as we mentioned in Section 3.1. For example, in `Mutable x (a, b, c)`, all the three fields of triple are independently mutable, while in `Mutable x (a, (b, c))` only two fields of type `a` and `(b, c)` are mutable—two fields of type `b` and `c` are not in toplevel and cannot be updated destructively. If we would like to make such updates possible, we would need rather complicated copying operations to keep further in-place updating safe. Suppose that state transformers of type `ST (a, (b, c)) _` could update all fields of type `b`, `c` and `(b, c)` respectively. This would be the case, if we provided both compositional references:

```
sndR' :: Mutable (x, y) y
sndR  :: Mutable (x, y) (Atom y)
```

Then we could update the field of type `(b, c)` with an immutable pair say, `(1, 2)` using `sndR`. Since fields `b` and `c` would be also updatable by using `sndR' `cmpR` fstR` and `sndR' `cmpR` sndR`, we would have to make a fresh copy of `(1, 2)` in order to make further updating safe. Otherwise, the data structure which should be immutable would be updated and this is, of course, wrong. However, if such copying is necessary, we can use `Mutable s (a, b, c)` instead. There is little reason to use `Mutable s (a, (b, c))`. On the other hand, we can use references in data structures (i.e. `Mutable s (a, Mutable s (b, c))`) in order to deal with *deeply* mutable data structures. Therefore, at least in most cases, it is sufficient to make only toplevel fields mutable and is reasonable not to provide compositional references of type `Mutable (a, b) b`.

Note that the framework of compositional references itself does not prohibit such compositional references. When in-place updating is not required, it is possible to provide both compositional references `sndR :: Mutable (a, b) (Atom b)` and `sndR' :: Mutable (a, b) b`, the latter of which makes (functional) update of deep fields without using references in data structures.

For recursive data types such as lists, the restriction above seems more problematic. We can not provide a compositional reference of type `Mutable [a] [a]`. Instead, we can provide only two compositional references of type `Mutable [a] (Atom a)` and `Mutable [a] (Atom [a])` for updating the head (car) and the tail (cdr) fields. Therefore, we can make *only the first* cons cell mutable. In order to make all the cons cells mutable, we must define the following new data type:

```
data MutList s a
          = MutCons a (Mutable s (MutList s a))
          | MutNil
```

One of the main reasons to introduce compositional references (`Mutable`) is to avoid introducing new mutable data types for corresponding immutable ones. However, we cannot obtain the type of mutable lists by just attaching `Mutable s` to the type of ordinary lists. In order to make automatic definition of the type of mutable lists possible, it might be necessary to change the form of recursive data type definitions as follows:

```
data List a = Cons a Self | Nil
```

where `Self` is a reserved word corresponding to the type being defined (`like Current` in Eiffel). Then, it would be possible to provide the mutable version of the type of lists (generally, recursive data types) automatically. Such modification would be also necessary for object-oriented style data type definitions where data types are defined incrementally as discussed in Section 3.3. Then, overloading would play a more important role in order to treat such families of data types uniformly.

6 Conclusion

We proposed a new type constructor `Mutable` and a set of new primitives (e.g. `appR`, `cmpR`, `fetch` and `assign`) based on the notion of compositional references. First of all, it enables us to use various space-efficient mutable data structures—otherwise, we would need to introduce primitive mutable data types in an ad-hoc manner. Second, when we do not need the expressiveness of the explicit style, we can write stateful programs in the implicit style. State transformers can use data structures such as arrays and tuples directly as state and have simple functional accounts and simple monomorphic types. Then they can be freely used as function parameters. We showed that such implicit style computations can be implemented efficiently by utilizing the relation between the implicit and the explicit styles.

Acknowledgments

The author would like to thank Atsushi Ohori and the anonymous referees for their helpful comments on earlier versions of this paper.

References

[1] Peter Achten, John van Groningen, and Rinus Plasmeijer. High level specification of I/O in functional languages. In Launchbury et al., editors, *Proceedings of Glasgow Workshop on Functional Programming*. Springer Verlag, 1993.

[2] Martin Hofmann and Benjamin Pierce. Positive subtyping. In *Annual ACM Symp. on Principles of Prog. Languages*, pages 186–197, 1995.

[3] Paul Hudak. Mutable abstract datatypes. Research Report YALEU/DCS/RR-914, Yale University Department of Computer Science, December 1992.

[4] Koji Kagawa. Mutable data structures and composable references in a pure functional language. In *Proc. of the Second ACM SIGPLAN Workshop on State in Programing Languages*, pages 79–94, January 1995. Technical Report UIUCDCS-R-95-1900, University of Illinois at Urbana-Champaign.

[5] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[6] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, June 1996.

[7] Eugenio Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, June 1989.

[8] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 65–77, January 1996.

[9] Frank Joseph Oles. Type algebras, functor categories, and block structure. In *Algebraic Methods in Semantics*, pages 543–573. Cambridge University Press, 1985.

[10] John Peterson, Kevin Hammond, et al. Report on the programming language Haskell version 1.3. `http://haskell.cs.yale.edu/haskell-report/haskell-report.html`, 1996.

[11] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Annual ACM Symp. on Principles of Prog. Languages*, 1993.

[12] Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. In *Annual ACM Symp. on Principles of Prog. Languages*, January 1993.

[13] John C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editor, *International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[14] Philip Wadler. Theorems for free! In *Proc. ACM Conf. Functional Programming and Computer Architecture*, pages 347–359, 1989.

[15] Philip Wadler. Comprehending monads. In *ACM Symp. on Lisp and Functional Programming*, pages 61–78, 1990.

[16] Philip Wadler. The essence of functional programming. In *Annual ACM Symp. on Principles of Prog. Languages*, 1992.

A Proof of the Lemma

Lemma: All the constants (*returnST*, *thenST*, *appR*, *fetch* and *assign*) satisfy the logical relations.

Proof:

We show the cases for *thenST* and *appR*. The other cases are easier.

(case for *thenST*)

Suppose that $m \sim_{ST\,\sigma\,\alpha} m'$ and $k \sim_{\alpha \to ST\,\sigma\,\beta} k'$. And suppose that

$$\begin{array}{l}\textbf{for any } s, g, q \\ s \sim_\sigma s' \textbf{ where } s' = rd\,q\,g\end{array} \qquad (17)$$

Let $(a, s_1) = m\ s$, $(a', g_1) = m'\ q\ g$ and $s_1' = rd\,q\,g_1$ then $(a, s_1) \sim_{(\alpha,\sigma)} (a', s_1')$ therefore $ka \sim_{ST\,\sigma\,\beta} k'a'$. From this we can conclude that $kas_1 \sim_{(\beta,\sigma)} (b', s_2')$ where $(b', g_2) = k'a'g_1$ and $s_2' = rdqg_2$. Moreover, from $g_1 = wrqg_2s_1'$, $g = wrqg_1s'$ and (3), we can conclude that $g = wr\,q\,g_2\,s'$.

(case for *appR*)

Assume that $r \sim_{Mutable\,\sigma:\tau} r'$ and $st \sim_{ST\,\sigma\,\alpha} st'$ and (17). We must show that $(r\ `appR^{`}_{cp}\ st)\ s \sim_{(\alpha,\sigma)} (a', s_1')$ and $g = wr\,q\,g_1\,s'$ where $(a', g_1) = (r'\ `appR^{`}_{nc}\ st')\,q\,g$, and $s_1' = rdqg_1$. From $s \sim_\sigma s'$:

$$\begin{array}{rcl} rd\,r\,s & \sim_\tau & rd\,r'\,s' \\ & = & rd\,r'\,(rd\,q\,g) \\ & = & rd\,(q\ `cmpR`\ r')\,g \end{array}$$

Then, it follows from $st \sim_{ST\,\sigma\,\alpha} st'$ that $(a, t_1) \sim_{(\alpha,\tau)} (a', t_1')$ and $g = wr\ (q\ `cmpR`\ r')\ g_1\ t'$ where

$$\begin{array}{rcl} t & = & rd\,r\,s \\ (a, t_1) & = & st\,t \\ (a', g_1) & = & st'\ (q\ `cmpR`\ r')\ g \\ & = & (r'\ `appR^{`}_{nc}\ st')\ q\ g \\ t_1' & = & rd\,(q\ `cmpR`\ r')\,g_1 \\ t' & = & rd\,(q\ `cmpR`\ r')\,g \end{array}$$

Then, we have to show that $wr\,r\,s\,t_1 \sim_\sigma s_1'$. For this, it is sufficient to show that $s_1' = wr\,r'\,s'\,t_1'$. To show this:

$$\begin{array}{rcl} \text{(r.h.s.)} & = & wr\,r'\,(rd\,q\,\underline{g})\,(rd\,(q\ `cmpR`\ r')\,g_1) \\ & = & wr\,r'\,(rd\,q\,\underline{(wr\,(q\ `cmpR`\ r')\,g_1\,t')}) \\ & & \quad (rd\,(q\ `cmpR`\ r')\,g_1) \\ & = & wr\,r'\,\underline{(rd\,q\,(wr\,q\,g_1\,(wr\,r'\,(rd\,q\,g_1)\,t')))} \\ & & \quad (rd\,(q\ `cmpR`\ r')\,g_1) \\ & = & \underline{wr\,r'\,(wr\,r'\,(rd\,q\,g_1)\,t')\,(rd\,(q\ `cmpR`\ r')\,g_1)} \\ & = & \underline{wr\,r'\,(rd\,q\,g_1)\,(rd\,r'\,(rd\,q\,g_1))} \\ & = & rd\,q\,g_1 \\ & = & \text{(l.h.s.)} \end{array}$$

In addition, we must show that

$$wr\,q\,g_1\,(rd\,q\,g) = g$$

To show this:

$$\begin{array}{rcl} \text{(l.h.s.)} & = & wr\,q\,g_1\,(rd\,q\,\underline{(wr\,(q\ `cmpR`\ r')\,g_1\,t')}) \\ & = & wr\,q\,g_1\,\underline{(rd\,q\,(wr\,q\,g_1\,(wr\,r'\,(rd\,q\,g_1)\,t')))} \end{array}$$

$$
\begin{aligned}
&= \underline{wr\ q\ g_1\ (wr\ r'\ (rd\ q\ g_1)\ t')} \\
&= \underline{wr\ (q\ `cmpR`\ r')\ g_1\ t'} \\
&= \text{(r.h.s.)}
\end{aligned}
$$

This completes the proof.