TUM

INSTITUT FÜR INFORMATIK

From Inheritance to Feature Interaction or Composing Monads

Christian Prehofer

TUM-I9715
April 97

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-04-I9715-300/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

From Inheritance to Feature Interaction or Composing Monads

Christian Prehofer,
Institut für Informatik,
Technische Universität München,
80290 München, Germany,
`www4.informatik.tu-muenchen.de/~prehofer`

June 3, 1997

Abstract

We show that techniques for monad composition can be used nicely for modeling object-oriented programming concepts. In this functional setting, we develop a new model for composing objects from individual features in a modular way. Features are similar to abstract subclasses, but separate the core functionality of a subclass from overwriting methods. We view method overwriting more generally as resolving interactions between two features. The interaction handling is specified separately and added when features are composed. This generalizes inheritance as found in object-oriented languages and leads to a new view of objects in a functional setting. Our concepts are implemented in Gofer and generalize some monadic programming techniques, where objects correspond to monads, features to monad transformers, and feature interactions are resolved by lifting functions through monad transformers.

1 Introduction

In this paper we model object-oriented programming concepts in a functional language and present generalizations of conventional object-oriented programming. Whereas the latter allows to develop classes of objects in an incremental manner, we just compose objects from a set of features, which replace classes. This approach was motivated by the recent interest in feature interactions in telecommunications, where service unit provides for of a set of (telephone) features. The crucial point is that some features may interact and have to be adapted in the presence of each other. This idea will be used for a novel approach to object-oriented programming. We consider such interaction handling for two features at a time and compose features with the appropriate interaction handling in a way which generalizes inheritance and method overwriting as in object-oriented programming. The flexible composition of features is achieved by advanced concepts of functional programming, in particular the monads and monad composition techniques. Our techniques allow to use object-oriented techniques while preserving the benefits of a higher-order lazy functional language, and also advance object-oriented programming concepts.

The feature model allows to compose objects from individual features (or abstract subclasses) in a fully flexible and modular way. Its main advantage is that objects with individual services can be created just by selecting the desired features, unlike object-oriented programming. A feature is similar to an abstract subclass and consists of a base implementation which

- adds functionality to an object
- may assume that the extended object provides other features.
- may add local state to the object (or may extend the used domains, e.g. by error cases)

Features are similar to abstract subclasses or mixins [5, 4]. The main difference is that we separate the core functionality of a subclass from overwriting methods of the superclass. We view overwriting more generally as a mechanism to resolve dependencies or interactions between features, i.e. some feature must behave differently in the presence of another one. For this purpose, we need to provide lifters, which adapt a feature to the context of another feature by overwriting methods. This leads to a new view of inheritance, as feature interactions are resolved between two features on a mutual basis. In contrast, inheritance just overwrites the method of the superclass.

The base functionality of a new feature is based on the functionality of the required ones and on the newly added state. This idea of assuming other features is a further difference to usual abstract subclass concepts. Note that the extended object can obviously have more than just the required features.

We use a modular architecture for composing features and the required interaction handling to a full object. As we only compose objects, there is no real notion of a class, which is hence often confused with the (type of) objects. The techniques we use for composing features have been developed for composing monads [20, 15] and have been used for handling interactions in interpreters for programming languages with several features [18, 9]. We program such feature interactions by lifting functions of one feature to the context of the other. This gives an architecture for composing features and interactions.

Whereas inheritance is used to extend a class with local state and functionality, we generalize this process and compose objects with individual services from a set of features. Although inheritance can be used for such feature combinations, all needed combinations, including feature interactions, have to be programmed explicitly. In contrast, we can (re)use features by simply selecting the desired ones when creating an object.

We claim that feature-oriented programming is advantageous for the following reasons:

- It yields more flexibility, as objects with individual services can be composed from a set of features. This is clearly desirable, if many different variations of one software component are needed or if new functionality has to be incorporated frequently.

- As the core functionality is separated from interaction handling, it provides more structure and clarifies dependencies between features. Hence it encourages to write independent, reusable code, as in many cases subclasses should be an independent entity, and not a subclass.

Our main technical contributions towards object- or feature-oriented programming are as follows.

- Using concepts for monad composition, we introduce a novel model for programming features in a modular and composable way which generalizes inheritance or subclassing.

- We show that some functionality (an undo function) which depends on several features can be implemented abstractly for any feature combination using type computations via type classes.

- We generalize some programming techniques used in [18] to generic classes of stateful and error monads.

An exposition of feature-oriented programming as an extension of an imperative language, namely Java, appears in [24]. This paper also includes a detailed comparison to object-oriented programming. Here, we focus the functional essence of this approach and on more advanced concepts, such as exception handling. This can also be viewed as semantical model of the core of the imperative version in [24].

We demonstrate our concepts by two examples, including some telecommunication features, where feature interactions have recently attracted great attention [27, 8]. For more examples in this area of telecommunications we refer to [25].

For implementing our concepts with monads we generalize techniques developed in [18]. In our model, classes correspond to monads, which can be viewed as particular abstract data types. The interesting point is that (some classes of) monads compose nicely and that we can build monad transformers, which transform an abstract data type to another. This is used to add features to objects. For instance, the mainly used monad transformers add (local) state (and extra functionality), from which we draw the comparison to inheritance. We show that implicit state via monads is essential for our abstract programming techniques. Similarly, overloading via type classes is important, as the type of polymorphic functions in feature implementations can only be determined after an object is composed from a set of features.

To compare this work with earlier results on monads, note that Moggi [20] aimed at lifting monads just by their types. This was extended to liftings for particular types of monads in [18], using their specific properties. Our technique is to name concrete instances of monad classes (e.g. state monads) and to program liftings depending on the names, but using generic liftings for the class of monads. As the names are identified with features, this clearly goes along the ideas of inheritance. Furthermore, we mostly use just state monads, which compose easily.

In the following, we present our concepts for writing features by an example, which will be the running example. Although we only use functions to access local variables of an object, the relation to object-oriented programming and to other concepts of inheritance should be clear. It is examined in detail in [24].

After a brief introduction to the technical concepts in Section 2, we show the concepts of stateful features in Section 3 and of error features in Section 4. The problems of multi-feature interaction are discussed in Section 5, followed by examples for stack features in Section 6. Another example for feature interactions in telecommunications is presented in Section 7.

1.1 A First Example

In the following, we show a small example modeling stacks with the following features:

(Basic) Stack, providing push and pop operations on a stack implemented by a list.

Counter, which adds a local counter (used for the size of the stack).

Undo, adding an undo function, which restores the state as it was before the last access to the object.

In an object-oriented language, one would extend a class of stacks by a counter and then extend this by undo. In general, a concrete class is added onto another concrete class. We will extend this to independent features which can be added to any object. For instance, we can run a counter object independently, or with undo.

The full implementation of the stack example contains six features which can be used modularly in many combinations.[1] It includes variations of the counter and the undo function. For instance, there is a version with a one-step undo and one with many-step undo. Another feature for handling stack underflow, based on a class of error features, is shown later in Section 4. We show in Figure 1

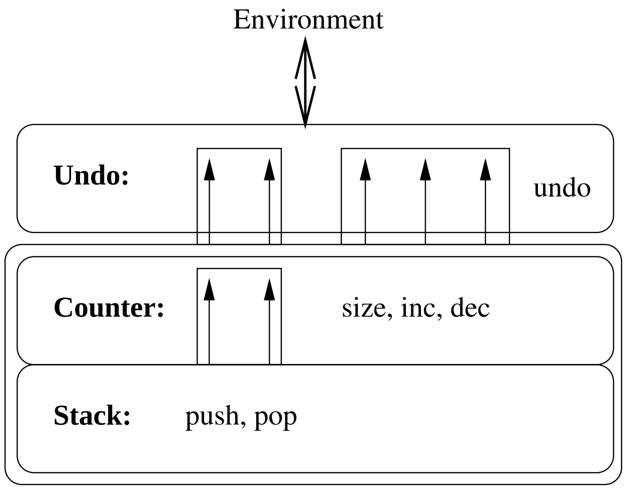

Figure 1: Composing features (rounded boxes) by lifters (boxes with arrows)

an example for feature composition with liftings, many more combinations are shown in Section 6. In this example we first add the counter to the basic stack. For this new object to support the stack feature, we have to lift the functions push and pop, indicated by arrows in the box denoting the lifting. This gives, like inheritance, a new object with two features, consisting of the inner two boxes. Since there are interactions between the two features, we must provide individual lifters for push and pop. Otherwise, one can use the default ones for composing orthogonal, independent features. With the undo component, we proceed similarly. Note that the functions push and pop are lifted again to undo, now with the lifter from stack to undo.

Clearly, these features are not independent. For instance, when adding the counter, the functions push and pop must, in addition, increment or decrement the counter. With traditional inheritance, this is achieved by overwriting of methods and by possibly calling the method of the superclass. In our setting, such dependencies are described by a lifting from one feature to a new context. Thus, liftings depend on two features.

To compose several features, liftings have to be more general: For any object having the set of features A, we can add feature b and lift the functions of each

[1] Code available via the autor's home page.

feature in A individually to the new context. Then we have an object which provides b as well. Using the structure of liftings, it is easy to model classical inheritance. Consider adding a feature b to an object with features A. To obtain a concrete subclass, one just has to merge the code of the feature a with all the lifters from $a \in A$ to b. Repeating this for all features, we can create a concrete class hierarchy for a particular object composed from some features. This amounts to the main difference to inheritance.

In the example, there are two lifters needed (two boxes) for adding undo to the object with counter and basic stack features. This is the main difference to inheritance, where a concrete class undo would extend a class with counter and stack and would redefine some of their functions. Whereas all this happens in one subclass, it is separated (and much more reusable) in three entities: one feature and two lifters.

Note further that lifting push and pop to undo does not depend on the counter; only the lifted versions of push and pop are lifted again by a lifter which depends on undo and basic stack.

We argue that liftings can nicely resolve many typical interactions between features, such as handling an extended local state. For instance, there is another interesting interaction between undo and counter. If a size request is followed by undo, shall the state before size or the one before the last push/pop request be restored? Such choices motivate a modular design, where not only the components are decoupled, but also their interaction. For instance, if the counter is not used, we do not want to bother with this complication.

1.2 Programming Features

To give a first idea of how to program features, we show (some of) the code for the stack and the counter features. Our concepts are provided by Gofer functions [13] and type constructions. We use the constructor classes of Gofer [14], which extend Haskell's type classes [21] and have been partly adopted in Haskell 1.3 [22].

We use monadic state transformers modeling implicit state as in imperative languages, which is essential for the desired flexibility and modularity. Composing features is done by the type system of Gofer with type constructions and type classes. A type class declares certain functions for its member types. Observe that type classes do not correspond to classes in object-oriented programming, but determine if a type has some feature. Thus a type can be in several type classes, vaguely reminiscent of multiple inheritance. Compared to object-oriented programming, type classes resemble the idea of interfaces, as e.g. in Java [10].

A type is in a type class (e.g. `StackMonad` or `CountMonad`) if the corresponding functions are provided in an instance declaration, as shown below. We use the type constructors $StackT, CountT$ to add features to a type. For instance, if m is the type of an object (a monad), then $StackT\ s\ m$ is a new type which also supports the stack feature with a local state of type s.

In the following code, the first type declaration for *StackT* declares that *StackT* is a state transformer, adding implicit state to the object of type m.[2] The second statement declares that *StackT* [*Int*] m is in the class *StackMonad* of stacks of integers.[3] (Note that [*Int*] is the type of lists over *Int*.) Furthermore, we have to give implementations for the functions which the feature provides, here push and pop. Note that we write types, type constructors and type declarations in italics.

```
  -- add implicit state of type s
  -- to m (simplified here)
type StackT s m  =  StateTrans s m

instance StackMonad (StackT [Int] m)
     where
   push a   = do{ s <- get;
                  put (a:s) }
   pop      = do{ s <- get;
                  put (tail s);
                  result (head s)}
   is_empty = do{ s <- get;
                  result (s==[]) }
```

In the above implementation, the do-notation for sequential computations in monads is used. Each statement in the do construct may compute a value and assign it to a local variable, e.g. `s <- get` assigns the result of `get` to `s`. In such a monad computation the added, implicit state can be modified via the functions `put` and `get`. Note that these access functions always refer to the implicit state of the "current" feature.

Next we show the counter feature, whose functions are also implemented via state transformers.

```
type CountT Int m  =  StateTrans Int m

instance CountMonad (CountT Int m)
       where
   size  = get
   inc   = do{ i <- get;
               put (i+1) }
   dec   = do{ i <- get;
               put (i-1) }
```

[2]State transformers will be explained in detail later. Also, the following type declaration is shortened. The full code and the class declarations are shown later.

[3]Polymorphic stacks are possible via a binary class `StackMonad`, using the extra argument for the type of the stack. However, this leads to ambiguous types later.

It remains to lift the functionality of stack to the context of a counter. The following instance declaration states that ($CountT\ Int\ m$) has the stack feature, under the preconditions (stated before the `=>`) that m has the stack feature, i.e. `StackMonad` m, and that ($CountT\ Int\ m$) is a `CountMonad`.

```
instance (StackMonad m,
       CountMonad (CountT Int m)) =>
       StackMonad (CountT Int m)
        where
   push a   =  do{ inc ;
                    lift (push a)}
   pop      =  do{ dec ;
                    lift pop}
```

The code for push first calls the increment function of the counter and then via `lift (push a)` the push function of the inner object ("superclass") of type m. Roughly speaking, `lift` corresponds to the function super as e.g. in Smalltalk and is, like `get` and `put`, defined later. Alternatively, if there is no interaction, one would just write

```
pop = lift pop
```

which could also be made a default (as implicit in object-oriented programming).

With the above code, an object of type

$$CountT\ Int\ (StackT\ [Int]\ m)$$

provides both features and behaves as expected. In general, liftings should preserve the functionality of the lifted features, i.e. an individual feature always behaves identically (if no others are used in between). For the standard lifting, this can be shown similar to [18].

The implementation of the undo feature is more involved and is presented in Section 5. The idea of the simple undo implementation is to save the local state of the object each time a function of the other features is applied (e.g. push, pop). The undo feature raises several new issues:

- The lifting of functions of the other used features is schematic: Always save the state first and then call the function to be lifted. In contrast to object-oriented programming, this can be done once and for all by a particular function

```
lift_undo f =
   do{ local_s <- lift gets ;
        put (Some local_s) ;
        (lift f) }
```

which lifts any function f to the undo feature. Note that lift gets refers to the state of the inner object.

- undo depends essentially on all "inner" features, since it has to know the internal state of the composed object. Since we work in a typed environment, the type of the state to be saved has to be known. This multi-feature interaction is solved by an extra feature, which allows to read and write the local state.

2 Monads, Type Classes and Features

In the following, we explain the technical background needed for our feature model. The ideas are based on investigations on features in programming languages [18]. The concept of monads has been introduced to programming for modeling state in functional languages [16] and for writing code which is easy to modify [26]. Both aspects will be essential in our context.

2.1 Type Classes

A type class in Haskell is essentially a set of types (which all happen to provide a certain set of functions). Each class declaration introduces a new class and a set of new function names, which are overloaded for each member of a class. For instance

```
class Eq a where
   eq ::  a -> a -> Bool
```

introduces the class Eq of all those types a which provide a function eq :: $a \to a \to Bool$. A class declaration is like a module interface: it separates declarations from implementations. Instance declarations provide the members of classes and concrete implementations for the member functions, e.g.

```
instance Eq Int where
  eq = eq_int
```

In general we can instantiate classes not just by base types but also by type terms. For example, we may wish to express that a type $[a]$ admits equality provided a does. This is achieved by the following instance declaration, where the Haskell notation => allows to add a list of type assumptions (here Eq a) for the new instance Eq $[a]$.

```
instance Eq a => Eq [a] where
  eq [] [] = True
  eq (a:as) (b:bs) =
     and [eq a b, eq as bs]
```

Note that the last two `eq` expressions refer to two different instances of `Eq`, one for a and one for $[a]$.

2.2 Constructor Type Classes

The extension to constructor classes of Gofer [14, 22] allows n-ary type classes. Furthermore, these arguments may not just be types, but can be type constructors. Let $*$ be the kind of types [3]. Then, for instance, the type constructor $[\,]$ (in mixfix notation) is of "kind" $* \to *$, as it maps types to types. Constructor classes are often used when standard type classes are too coarse to describe the types of the member functions. The standard example is the binary container class, whose instances typically are lists and trees:

```
class Container c a where
  member :: a → (c a) → Bool
```

Here we can express that the type $c\ a$ depends on a. If $c\ a$ is replaced by a type s, in a `class Container' s`, then the type of `member ::` $a \to s \to Bool$ would be too general: we cannot write a sensible function which for any type a checks membership in a type s. Typical instance declarations are:

```
instance Container List a where
  member e []  = False
  member e a:s = or [eq e a,
                     member e s]

data Tree a   = Leaf a
              | Node (Tree a )
                     (Tree a )

instance Container Tree a where
  member e (Leaf a)   = eq e a
  member e (Node a b) =
     or [member e a, member e b]
```

2.3 Monads

Programming with monads provides a compromise between imperative languages, where statements affect an implicit, global state, and stateless functional languages, where all information flow is — sometimes tediously — explicit. Monads also separate building computation (e.g. composing state transformers) and running a computation.

A monad is a type constructor m with some operations and laws. If a is a type, then $m\ a$ is the type of a larger object which "wraps" a, often a function type (e.g. a state transformer) as shown later. In monadic style, a function from a to b is assigned the type $a \to m\ b$. There are standard functions to work with monads, defined in the type class for monads, which builds upon the functor class:

```
class Functor m where
 map :: (a → b) → (m a → m b)

class Functor m => Monad m where
 result :: a → m a
 bind   :: m a → (a → m b) → m b
```

Function `result` inserts a value into the "empty" monad and `bind` applies a monadic function to a value of type `m a`. Note that we use the do-notation for `bind`, defined as

```
 do { x <- m ; t} =def m bind λx.t
```

This notation extends canonically to sequences of bind applications. The monad laws for `result` illustrate the "empty" monad:

```
 (result a) bind λb. t = [a/b] t
  m bind λb. result b  =  m
```

where `[a/b]` is a substitution mapping `b` to `a`. (See [26] for more details on monad laws.)

2.4 Features: Monads with Operations

Features are defined as monads with additional operations. These can be viewed as predicates over types which characterize the features. For instance, for the basic stack and counter features we define:

```
  -- type of stack elements
type St = Int

class  Monad m =>
     StackMonad m  where
  push     :: St → m ()
  pop      :: m St
  is_empty :: m Bool

type Ct = Int  -- type of counter

class Monad m =>
```

```
    CountMonad m  where
  size      ::  m Ct
  inc       ::  m ()
  dec       ::  m ()
```

This declares the two classes used in the introduction, `StackMonad` and `CountMonad`, with their corresponding functions. It assumes that `m` is a monad. (Note that () is the empty type.)

3 A Class of Stateful Monads

We show in the following the underlying machinery for features which add state to some object. The basis of state monads is a type

`type` $StateTrans\ s\ m\ a\ =\ s \rightarrow\ m(s,\ a)$

which extends any monad m to a type of a state transformer for a state of type s. This transformer can be applied repeatedly, i.e. $StateTrans\ s\ m$ is again a monad, as shown below. For the following general model, we generalize over this type and just assume the functions `closeS` and `openS`. These access the internal structure of state monads and are only used internally.

The ternary class `StateMonadT` $c\ s\ m$, where s is the type of the added state, m a monad and c an appropriate type constructor, declares that $(c\ s\ m)$ is a stateful monad with the following functions (for some of which definitions are included):

```
class Monad m =>
    StateMonadT c s m  where
 closeS ::(s -> m(s, a)) -> c s m a
 openS  ::c s m a -> s -> m(s, a)

 get    :: c s m s
 get    = closeS(λs.result(s,s))

 put    :: s -> c s m ()
 put a  = closeS(λs.result(a,()))

 lift   ::  m a -> c s m a
 lift m = closeS(λs. do{
           x <- m; result(s ,x)})
```

For the functions `get`, `put` and `lift`, also definitions are provided in the class declarations. The functions `closeS` and `openS` are used to show that any state monad is a monad:

```
instance  StateMonadT c s m =>
   Functor (c s m)  where
   map f xs = closeS (
       λs. (openS xs) s bind
       λ(s',x). result(s', f x))

instance  StateMonadT c s m =>
    Monad (c s m)  where
    result x = closeS(
                λs.result(s,x))
    m bind k = closeS(
        λs0. (openS m) s0 bind
        λ(s1, a). openS (k a) s1)
```

This generic class generalizes the various stateful monads in [18], where the above definition of monads is repeated for stateful monads.

3.1 Defining a Stateful Feature

With the above concepts, we can show in detail the definition of basic stack features. Only the following data type declaration is needed,[4] as well as declaring it to be a stateful monad.

```
data StackT s m a =
        STM(StateTrans s m a)

instance StateMonadT StackT s m where
   closeS  x       = STM x
   openS (STM x)   = x
```

Similar declarations are needed for the counter feature. The instance declarations for *StackT* and *CountT* can be found in Section 1.2.

4 A Class of Error Monads

As for stateful monads, we can similarly define a generic monad which adds extra values to the computation. For instance, with the above definition of stacks, stack underflow results in a program error. Using error monads, we can cope nicely with such cases. In applications it is then possible to use stacks with or without error handling as needed.

Whereas stateful monads build upon a particular function type (*StateTrans*), we use a sum type here:

[4] Note that we use an extra constructor `STM` to define *StackT* via a data type definition. This is needed for type checking.

```
data Err e a    =  Data a | Error e

type ErrT e m a  =  m(Err e a)
```

Thus $ErrT$ adds error elements of type e to a monad m. Note that this composes with state monads. For instance, we obtain the type

$$(ErrT\ \ e\ \ (StackT\ \ s\ \ Id))\ \ a\ \ =$$
$$STM(\ s\ \ \rightarrow\ \ Id(s,\ \ Err\ \ e\ \ a))$$

The class of error monads supports open and close functions as for state monads, plus generic functions to inject and check errors (`put_err, read_err`), and the canonic lifting function `lifterr`.[5]

```
class Monad m =>
   ErrMonadT c s m where
 openE    :: c s m a  ->  m(Err s a)
 closeE   :: m(Err s a)  ->  c s m a

 put_err  :: s  ->  c s m a
 read_err :: c s m a  ->  c s m Bool
 lifterr  :: m a  ->   c s m a
 lifterr c = closeE (map Data c)

 put_err  s =
     closeE(result(Error s))
 read_err m =
     closeE(map isError (openE m))
  where
    isError (Error s) =  Data True
    isError (Data x)  =  Data False
```

Showing that `ErrMonadT` $c\ s\ m$ is a monad is more complicated. It can for instance be shown if we assume that m is any StateMonad. For this we use the concepts of [15], which can be generalized to classes of monad transformers.

For instance, an error handler for stack underflow is written by lifting stack over Err, using Int for error values. Since we only use the base functions of `ErrMonadT`, we don't need to introduce an extra class and a type constructor for this. (An example with an explicit class is shown in Section 7.2.)

```
instance (StackMonad m ,
```

[5]Due to the type system, the function cannot be overloaded to work under the same name as in stateful monads. Adding an extra class for monad transformer is no solution, as typing does not permit to declare instances for both classes of monads.

```
          ErrMonadT ErrT Er m)=>
      StackMonad (ErrT Er m)
        where
   pop      = do{
           b <- is_empty ;
           if b then (put_err 0)
                else (lifterr pop)}
   push a   =  lifterr (push a)
   is_empty =  lifterr is_empty
```

Lifting other, independent features is canonical:

```
instance (CountMonad m,
          ErrMonadT ErrT s m) =>
        CountMonad (ErrT s m)  where
   size = lifterr size
   inc  = lifterr inc
   dec  = lifterr dec
```

This lifting can even be generalized to any state monad, if `CountMonad` is independent of all other stateful features.

In the current model for features, we have just provided generic monad composition for a set of stateful features with one error feature. Although it is possible to use several error features, it is easier to use one error monad transformer and to build other features on top of it. For instance, we only use the integer 0 as error message here and leave others open for other error cases. (In case several features use the same error message, we can treat this as an interaction.)

5 The Undo Feature: Multi-Feature Interaction

We continue the stack example by introducing the undo feature, which has interesting interactions with several other features. The problem is that the undo feature must access the local states of all (stateful) features the object already has. Since we work in a typed setting, we also need the type of all local states. Hence undo depends on several features. As we work with standardized monads, it is possible to add an auxiliary feature,[6] which determines the state of an object and provides access to it. Thus undo can be added to any feature combination.

The additional class `SMonad` for stateful monads is declared via

```
class Monad m => SMonad s m where
  gets  :: m s
  puts  :: s -> m s
```

[6] Not shown in Figure 1.

This binary class declares that monad m has state s and provides access functions. Instances can be defined schematically for both classes of monads, e.g.:

```
instance (SMonad s0 m,
          StateMonadT c s m) =>
         SMonad (s,s0) (c s m)
          where
 gets       = do{s  <- lift gets  ;
                 s' <- get ;
                 result (s',s) }
 puts (a,b) = do{s <- lift (puts b);
                  put a }
```

This expresses that $c\ s\ m$ has state $(s, s0)$, if m has state $s0$. Now we can define the undo feature via `SMonad` as follows. Since there may be no saved state for undo, we use the data type *Option* for the copy of the local state in the following code:

```
data Option a  =  Some a | None

data UndoT s m a  =
    UTM(StateTrans s m a)

instance StateMonadT UndoT s m where
  closeS  x     = UTM x
  openS (UTM x) = x

class  Monad m => UndoMonad m where
  undo   ::  m ()

instance  SMonad s m =>
     UndoMonad (UndoT (Option s) m)
       where
   undo    =  do{
              u <- get ;
              case u of
               None    -> result ()
               Some u1 ->
                   lift (puts u1)}
```

The other interesting point about undo is lifting of functions of other features. The advantage is that lifting proceeds via the following generic scheme, which first extracts the local state of the object, updates the saved state and then calls the lifted function:

```
liftundo  f =
  do{local_s <- lift gets ;
      put (Some local_s) ;
      (lift f) }
```

Lifting for the basic stack features proceeds canonically:

```
instance (SMonad s0 m,
          StackMonad m ) =>
      StackMonad (UndoT (Option s0) m)
        where
  push a = liftundo (push a)
  pop    = liftundo pop
```

There is an interesting interaction when the counter is used. For lifting `size`, which does not affect the state, we can either overwrite the saved state or leave it unchanged (as shown in the comment in the code below). In the former case, `undo` after `size` will have no effect. With our model of feature interaction, we just have to use the appropriate lifting function for interaction resolution.

```
instance (SMonad s m,
          CountMonad m ) =>
      CountMonad (UndoT (Option s) m)
      where
    size          =  liftundo size
 -- alternative:  =  lift size
    inc  = liftundo inc
    dec  = liftundo dec
```

Currently, just one lifting between two features is possible due to the type system. A further step would be to allow more liftings and to parameterize over liftings.

6 Using the Stack Features

A simple example for an object (monad) with two features is the following, which uses the identity monad Id with no features as base monad. By the following type declarations features are selected.[7] Running the above state transformers requires extra machinery for injecting an initial state and for extracting the computed value.

```
  -- stack with counter
```

[7] Gofer can infer the types without these declarations, but the inferred type is too general, as Gofer allows several (base) implementations for a type class.

```
test1 :: (CountT Ct (StackT [St] Id)) St
test1 =   do{
  push 1 ;
  push 2 ;
  size }     -- computes 2

  -- stack with undo
test2 ::  (UndoT (Option ([St],()))
          (StackT [St] Id)) [St]
test2 =  do{
  push 1 ;
  push 2 ;
  push 3 ;
  undo ;
  p2 <- pop ;
  undo ;
  p1 <- pop ;
  result [p1,p2]} -- computes [2, 2]

-- stack with counter + undo
test3 ::  (UndoT (Option (Ct, ([St],())))
      (CountT Ct (StackT [St] Id))) [St]
test3 =  do{
  push 1 ;
  push 2 ;
  push 3 ;
  undo ;
  p2 <- pop ;
  s <- size ;
  p1 <- pop ;
  result [p1,p2,s]}
    -- computes [1, 2, 1]

--  counter with undo
test4 :: (UndoT (Option (Ct, ()))
          (CountT Ct Id)) St
test4 =  do{
  inc;
  inc;
  undo;
  size }              -- computes 1
```

7 Feature Interaction in Telecommunications

In telecommunications, feature interaction problems have led to a new research branch [27, 8] focusing on such interaction problems which hinder the rapid creation of new services. The problem in feature interaction stems from the abundance of features telephones (will) have. For instance, consider the following conflict occurring in telephone connections: B forwards calls to his phone to C. C screens calls from A (ICS, incoming call screening). Should a call from A to B be connected to C? In this example, there is a clear interaction between forwarding (FD) and ICS, which can be resolved in several ways. For many other examples we refer to [7].

We demonstrate our techniques, including an example for virtual functions, with the following set of features for this domain of connecting calls:

- ICS (incoming call screening)
- Forwarding of calls
- Error handling for busy phones (also used for disallowed calls)

The first two of these features add local state, i.e. the origin of the call, which is not needed for the other features.

In this application, there are similar feature interactions as in the last section. The interactions mostly stem from extending the environment or from resource conflicts. The first can be handled by liftings, the second by the order on features.

Our full implementation contains another feature, called OCS (outgoing call screening), which is similar to ICS. Already with four features and several resolutions to the interactions, there are many different feature combinations.

7.1 Forwarding

The goal in the following is to provide functionality for connecting calls.

```
  -- type for phone numbers
type Dn = Int

class   PMonad m =>
     FWDMonad m where
  forward :: Dn -> m Dn
```

Forwarding only uses two (constant) lookup functions `fd_check` and `fd` with forwarding information and adds no local state. For simplicity, we use a state transformer which adds no state.

```
data FwdT s m a =
  FTM (StateTrans () m a)

instance StateMonadT FwdT () m where
   closeS  x     = FTM x
   openS (FTM x) = x

instance FWDMonad (FwdT () m) where
  forward nr =
    if (fd_check nr)
        then result (fd nr)
        else result nr
```

7.2 The Busy Monad

The Busy monad provides a function for raising a busy signal and is based on the error monad.

```
class Monad m => PMonad m where
  raise_busy :: m a

type PhoneT = ErrT ()

instance ErrMonadT ErrT () m =>
      PMonad (PhoneT m) where
  raise_busy = put_err ()
```

7.3 Incoming Call Screening

For ICS we use a state monad with the origin of the call as local state:

```
data IcsT m a =
  ITM (StateTrans Dn m a)

instance StateMonadT IcsT Dn m where
  closeS  x     = ITM x
  openS (ITM x) = x

class IcsMonad m where
  check_ics  :: Dn -> m Dn
```

The corresponding implementation uses a function `check_ics1`, which checks disallowed callers:

```
instance IcsMonad (IcsT Dn m) where
check_ics dest   =  do{
     orig <- get;
     if (check_ics1 orig dest)
         then result dest
         else raise_busy }
```

7.4 Resolving the ICS/Forward-Interaction

To resolve the interaction between forwarding and ICS, we lift the forward function to ICS. If we choose the standard lifting by

```
instance (FWDMonad m,
          StateMonadT IcsT a m) =>
   FWDMonad (IcsT a m)  where
 forward  a = lift (forward a)
```

then the local state added by ICS is not affected by forwarding. Hence, the ICS check uses the origin of the call. If the intermediate hop is to be used, we would write

```
 forward a = do{put a;
                lift (forward a)}
```

instead. Note that `get` and `put` refer to the ICS feature here. Again, lifting allows a modular resolution of the interaction between two features.

8 Conclusions and Related Work

We have presented a novel model for feature-based programming where features can be defined individually and are separated from interactions with other features. This is the main difference to other concepts of abstract subclasses or inheritance. Thus it is much more flexible and has a larger potential for reuse.

We have shown that the architecture of monad compositions is suitable for typical feature and interaction handling. It should be noted that we use monads mostly to provide an abstract interface to implicit state. Apart from this, our composition techniques are essentially just composition of abstract data types, for which we use type classes. This, however, does not hold anymore if other "programming features", e.g. error handling, are involved.

Note that we only construct one object from some set of features. Using several objects can be done by some model of object identifiers (as for instance in [17]). This, however, is orthogonal to the feature model. Modeling a global object store with monads is possible, but the type system of Gofer cannot express all the needed construction nicely.[8] For this extension, dependent types would

[8] A Gofer program is available from the author.

be useful, as an object should have information about the type of its instance variables, which are maintained in a global store. Hence, we currently work on formalizing this using LEGO [19].[9]

Another extension to our presentation are virtual methods with late binding. Using virtual methods in a feature can just be seen as an assumption on the full object, which is composed of several features. When creating an object, this assumption can be discharged. As this requires to have a notion of objects, it is practical to model this with a global object store, as discussed above.

Type classes provide for a nice implementation, but do not fully match our programming concepts. First, we generally assume an interface (or class) definition for a feature with just one concrete base implementation plus several liftings, whereas type classes would allow for more implementations.[10] Furthermore, some features cannot be made polymorphic, as the Gofer type class system requires all type variables in the parameters of a class to appear in the type declarations of member functions.

Another approach to model subclassing and inheritance with type classes was presented in [12]. In this extension of Haskell, classes can be defined by extending (or reusing) other classes, but the work does not go beyond the concepts of object-oriented languages.

Compared to the modular interpreter in [18], we develop a concept of features on the language level, instead of describing semantics of a programming language. Furthermore, we generalize the programming techniques used in [18] and also address the problem of dependencies between several features. For the model of features, we also need the idea of assuming certain other features, as shown above. In earlier works [16, 26], monads are used to write easy to modify code with stateful features. We go the step beyond and write easy to configure components. In other words, we make the possible modifications explicit.

Type theoretic approaches, e.g. [2, 1, 23], aim at modeling object-oriented phenomena, but not at features. The essential difference is that features are designed such that we can add a feature to any object which supports the required other features.

Acknowledgments. The author is grateful to W. Naraschewski and to M. Broy for discussions and to the latter for suggesting the undo example.

References

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. In *European Symposium on Programming (ESOP), Edinburgh, Scotland*, 1994.

[9] Jointly with W. Naraschewski

[10] Clearly, liftings are also an implementation using overloading. But these are usually used in a restricted fashion and assume an underlying implementation.

[2] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, 1994.

[3] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

[4] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, San Francisco, April 1992. IEEE Computer Society.

[5] Gilad Bracha and William Cook. Mixin-based inheritance. *ACM SIGPLAN Notices*, 25(10):303–311, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[6] K. Brockschmidt. *Inside OLE (2nd Ed.)*. Microsoft Press, 1995.

[7] E.J. Cameron, N. Griffeth, Y.-J. Linand M.E. Nilson, W.K. Schnure, and H. Velthuijsen. A feature interaction benchmark for in and beyond. In L. G. Bouma and Hugo Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 1–23, Amsterdam, 1994. IOS Press.

[8] K. E. Cheng and T. Ohta, editors. *Feature Interactions in Telecommunications III*. IOS Press, Tokyo, Japan, Oct 1995.

[9] D. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, September 1996.

[11] The Object Management Group. *Common Object Service Specification*. John Wiley & Sons, 1994.

[12] J. Hughes and J. Sparud. Haskell++: An object-oriented extension of Haskell. Haskell Worksop 1995, available via `http://www.md.chalmers.se/~sparud/`.

[13] Mark P. Jones. Introduction to Gofer 2.20. Technical report, Yale University, September 1991.

[14] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), January 1995.

[15] Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report RR-1004, Yale University, December 1993.

[16] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, January 1993.

[17] Samuel N. Kamin and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. The MIT Press, 1994.

[18] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *22nd ACM Symposium on Principles of Programming Languages*, San Francisco, California, 1995.

[19] Zhaohui Luo and Randy Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[20] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[21] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *J. Functional Programming*, 5(2):201–224, 1995. Short version appeared in POPL '93.

[22] John Peterson and Kevin Hammond et al. Report on the programming language Haskell: A non-strict, purely functional language. Technical report, May 1996. Version 1.3.

[23] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title “Object-Oriented Programming Without Recursive Types”.

[24] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP '97*, 1997. To appear in Springer-LNCS.

[25] Christian Prehofer. An object-oriented approach to feature interaction. In *Fourth IEEE Workshop on Feature Interactions in Telecommunications networks and distributed systems*, 1997. to appear.

[26] P. Wadler. Monads and functional programming. In M. Broy, editor, *Proceedings of the 1992 Marktoberdorf international summer school on program design calculi*. Springer Verlag, 1993.

[27] P. Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, XXVI(8), August 1993.