From Inheritance to Feature Interaction

Christian Prehofer*

— *Extended Abstract* —

1 An Approach to Modular Feature Composition

In this paper we introduce a new model for composing objects from individual features in a fully flexible and modular way. Its main advantage is that different objects can be created just be selecting the desired features. In addition, there may be dependencies or interactions between features, i.e. some feature must behave differently in the presence of another one. We present a modular solution to such feature interactions by lifting functions of one feature to the context of the other. This extends the idea of overwriting methods in inheritance, but is modular.

We will introduce the model by comparing it to inheritance, although it is not our goal to extend inheritance with all its ingredients from object-oriented programming languages. Instead, we view inheritance as a mechanism to derive new modules from old ones.

First we need to clarify the basic concept: a feature (or component)

- adds functionality to any object which provides
- some required other features and
- may add local state to the object (or extend the used domains, e.g. by error cases)

Clearly, for providing the new functionality, the new feature can only use the functionality of the required ones and the newly added state. Furthermore, lifting of other features is needed, as shown below.

Compared to inheritance, new objects can be constructed just by telling which features to use. Although inheritance can be used to solve such feature combinations, all needed combinations have to be programmed explicitly. With additional feature interactions, matters get even more involved.

*Fakultät für Informatik, Technische Universität München, 80290 München, Germany, prehofer@informatik.tu-muenchen.de

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

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

Counter, which adds a local variable counting the number of elements in the stack.

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

Clearly, these features have dependencies. For instance, when adding the counter, the functions push and pop must, in addition, increment or decrement the counter. With classical inheritance, this is achieved by overwriting of methods and by possibly calling the method of the superclass (e.g. via `super`). 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, lifting have to be more general: For any object having feature A, we can add feature B and lift the functions of A to the new context. Then we have an object which provides both features.

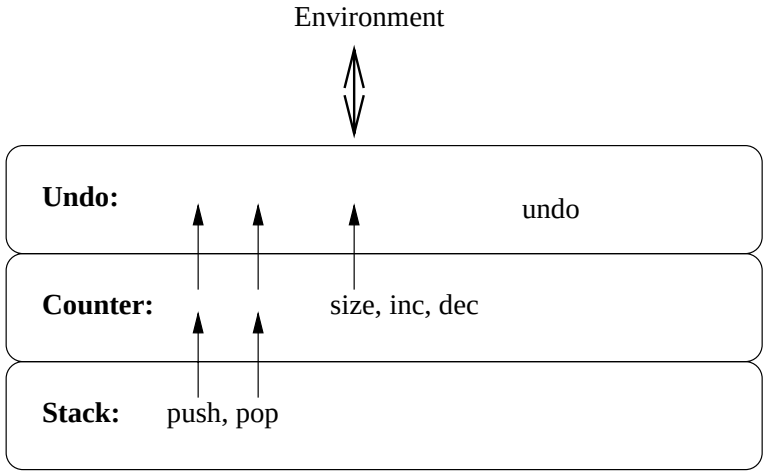

Figure 1: Composing Stack Features (Arrows denote lifting of functions)

In Figure 1, we show the structure of feature composition and liftings. To the basic stack, we first add the counter. For this new object to support the stack feature, we have to lift the functions push and pop, indicated by arrows. This gives a new object with two features, consisting of the lower two boxes. Since there are interactions, 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. Observe that the functions push and pop are lifted again. There are few more subtle issues, as discussed in the following

section. For instance, the inc and dec functions are not lifted, which corresponds to hiding.

The main advantage in our setting is that features can be configured (almost) arbitrarily. For instance, we can omit the counter or provide an alternative counter with different implementation or additional functionality. Also for the undo-component, different versions are available: one with a one-step undo, one with many-step undo. In addition to the ones shown above, there is also an error handling feature for stack underflow, based on a different class of monads.

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. Furthermore, if the Counter is not used, we do not want to bother with this complication. In traditional object-oriented programming, one could either build separate versions of Undo (e.g. one with Counter and one without), or a monolithic one considering all interactions. With respect to flexibility and reusability, both alternatives are not satisfactory.

On the technical side, we implement our concepts by monads, inspired by techniques developed in [7]. In our model (functional) objects correspond to monads, which can be viewed as particular abstract data types. The interesting point is that monads compose nicely and 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 analogy to inheritance. For the used classes of monads, generic introduction of new components and composition concepts are provided.

2 A Glimpse of the Implementation

To give a rough idea of the implementation (done in the functional language Gofer [5]), we show the main code fragments.[1] Since this is just a first prototype encoding of the ideas, our concepts are provided by Gofer functions and type constructions. Furthermore, we use explicit state transformers in contrast to implicit state in imperative languages. Although this is a bit clumsy in our first implementation without any syntactic sugar, it is essential for the desired flexibility and modularity.

Composing features is done by the type system of Gofer [6] with type constructors and type classes [8]. Adding features is done by type constructors. 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, somewhat similar to multiple inheritance. (Note that we often do not distinguish between objects and their type.)

A type is in a type class (e.g. `StackClass, CountClass`) if it provides the corresponding functions. Furthermore, we use the type constructors `StackT, CounterT` to

[1] Slightly shortened and polished. The full code is available by the author.

add features to a type. If `m` is the type of an object, `(StackT m)` adds the feature Stack to it. In the following code, the first type declaration for `StackT` declares that `StackT` adds a local state, which is a polymorphic list over a type `a`. The second statement declares that `(StackT m)` is in class `StackClass` and provides implementations for push and pop.

```
type StackT = StateTransformer (List a)

instance StackClass (StackT m) where
   push a    = update ( \stack -> (a:stack,a))
   pop       = update ( \stack -> (tail stack, head a))
```

In the implementation, the added state can be modified via the function `update`, which applies its functional parameter to the internal state. (It thus constructs a state transformer.) For instance, to implement push, we use the function `\stack -> (a:stack,a)`, where `\x ->` denotes functional abstraction over x and `a:stack` appends the list `stack` to the new element `a`. Note that the function must return a pair, here `(a:stack,a)`, where the first element is the new state and the second the returned value (here the return value of push). For instance, pop returns the head of the list and removes the first element from the stack by assigning `tail stack` to the local stack.

Next we show the Counter feature, whose functions are also implemented via state transformers using the `update` function. Note that `size` leaves the counter `i` unchanged and returns its value.

```
type CountT = StateTransformer Int

instance CountClass (CountT m) where
   size  = update (\i -> (i,i))
   inc   = update (\i -> (i+1,i))
   dec   = update (\i -> (i-1,i))
```

It remains to lift the the functionality of Stack to the context of a counter. The following code states that if `m` has feature Stack, i.e. `StackClass m`, then `(CountT m)` also has the Stack feature.

```
instance StackClass m => StackClass (CountT m) where
   push a   =  inc `bind`  lift (push a)
   pop      =  dec `bind`  lift  pop
```

In the code, we use a particular infix operator `bind`, which composes state transformers and roughly corresponds to the semicolon in imperative languages. 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 ("super class") `m`. Roughly speaking, `lift` corresponds to the function super as e.g. in Smalltalk and is, like update, provided in the system.

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 (StackT m)` provides both features and behaves as expected.

The implementation of the Undo feature is not shown here for lack of space. The idea is 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 involves several interesting new issues:

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

  ```
  lift_undo f = save_state `bind` (lift f)
  ```

  lifts any function `f` to the Undo feature.

- 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 problem is solved by extra functionality of state transformers, which allows to read and write the local state.

- As shown in Figure 1, some functions of Counter are not lifted, corresponding to hiding.

In the current implementation, we have six components. All can be added modularly in many combinations and their interactions are decoupled and easy to maintain.

3 Feature Interaction in Telecommunications

In the telecommunications and multimedia development, feature interaction problems have led to a new research branch [10] 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. This was in fact the original motivation for this work. 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 the components for Forwarding (FD) and for ICS, which can be resolved in several ways. For many other examples we refer to [3].

We have implemented a set of features for this domain of connecting calls:

- ICS (incoming call screening)

- OCS (outgoing call screening)

- Forwarding of calls
- Error handling

In this application, there are similar feature interactions as in the last section. Already with four features and several resolutions to the interactions, there are more than a dozen different configurations.

This application area also seems more suitable for this approach: Different configurations and the addition of novel features must be handled. Furthermore, the interactions mostly stem from extending the environment or from resource conflicts.

4 Related Work

In the above, we already compared our approach with inheritance. In object-oriented programming, there are several other concepts not addressed here. Some of them, such as virtual methods with late binding, are also possible here, but others, like subtyping and dynamic binding are not yet considered.

Other type theoretic approaches, e.g. [2, 1, 9], aim at modeling object-oriented phenomena, but not at features. The essential difference is that we design in such a way that we add a feature to *any* object which supports the required other features and if lifters are provided.

Another motivation for this work is the current trend in commercial software, componentware, which is driven by the idea to compose small, reusable software components to larger ones. In industrial applications, componentware is largely successful by standardizing component platforms, architecture and interfaces, as e.g. demonstrated by CORBA and SOM. Although the idea of simple plug-and-play with software components is tempting, it is a bit simplistic. Even when syntactic interfaces are compatible, a typical problem is that interaction or conflicts occur between individual components, as shown here.

5 Conclusions

The main goal of component-based or feature-based programming is to specify and implement features of components and their interactions separately. Towards this goal, have shown that feature composition can be fully modular, also with respect to interactions. Clearly, the interactions are not deep algorithmic dependencies, but cover many practical problems.

Not mentioned are the extra new capabilities of our model. For instance, we can also used different monads, called ErrorMonads, which add error cases (or any other special cases) to objects. For instance, it is possible to lift functions over undefined values and to provide (modular) error handling. Functions, such as raising errors, can be lifted through features.

6 Relation to the Workshop

We have presented a first approach to a new, general concept of features, which goes beyond inheritance. The modular structure of this approach allows to disassemble and to pinpoint the problems of classical inheritance with overwriting of methods. For instance, in each of the two case studies, we have implemented only four to six features, but there is already an abundance of possible configurations and interactions. Only with such a modular design, we can hope for flexible and reusable software. Thus this approach addresses two main problems with inheritance:

- Inheritance breaks modularity by overwriting methods.
- Inheritance is too rigid and inflexible, since individual (sub)classes cannot be (re)used without a detailed understanding of the class hierarchy.

Another, somewhat orthogonal, approach to the last problem is the idea of design patterns [4], aiming at standardized class structures. This does however not address the composition problems solved here.

References

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

[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] 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.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Micro-Architectures for Reusable Object-Oriented Design.* Addison Wesley, Reading, MA, 1994.

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

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

[7] 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.

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

[9] 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".

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