

Objects and Subtyping in a Functional Perspective

Martin Odersky

IBM Research, T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, N.Y. 10594
odersky@yktvmh.bitnet
Tel: (914) 784 - 7960

March 20, 1992

Abstract

Object oriented programming languages with subtyping facilitate the re-use of code which forms the top part of a call-graph. This style of code reuse, sometimes called inverted programming, cannot be simulated easily in traditional languages. We present a technique for achieving the same effect in the framework of a higher-order, polymorphic functional language without subtyping. Our method establishes a close relationship between the object-oriented concepts of class, “self”, and subtyping and the functional concepts of algebraic data types and abstraction functions. The relationship can serve as a transformational semantics of object-oriented language elements.

Keywords: Code Reuse, Subtyping, Parametric Polymorphism, Haskell, Abstraction Functions.

Topic: Research

4600 Words

1 Introduction: Facets of Code-Reuse

The biggest advantage of using an object-oriented programming language instead of a conventional one is the increased ability of reusing code. This work was motivated by the question whether object-oriented techniques supporting code reuse can be translated into the framework of a polymorphic, higher-order functional language without subtyping.

We identify two forms of code reuse. First, common patterns of data and code can be collected in in a class. Descendants of the class automatically inherit these patterns. Thus, only new and changed functionality has to be coded explicitly. Liskov [14] argues that the same effect can in many circumstances be achieved by a server/client relationship between abstract data types corresponding to a class and its descendants.

But there is a second form of code reuse which cannot be simulated so easily. Traditional function libraries allow code reuse only if the reused portion of code is invoked by the new code; hence, only the bottom part of the call graph can be reused. This severely restricts our ability to factor out common behavior. A prime example is found in the area of simulation where the invocation of objects, central time keeping and event handling remain fixed, but the behavior of individual agents should vary according to the simulation at hand. Traditional techniques of program composition do not support this style of reuse; the only way to work around the problem is to re-code (or at least re-compile) the central functions in every simulation program, or to use a special simulation notation such as GPSS. The design of a special language for object behavior amounts to completely removing the bottom part of the call graph by transforming it into a data structure. The reusable top part then acts as an interpreter which evaluates this structure. It comes as no surprise that the first general-purpose object-oriented programming notation, Simula, was designed with precisely the problem of simulation in mind.

Another example of reusing the top of the call graph is represented by a window system. Its central part consists of a loop which collects events from sources such as the keyboard or the mouse and determines the identities of the window which should react to the current event. Each window acts as a command interpreter of its own. The particular kind of action taken upon an event is not known to the central dispatcher, nor are the types of data structures used during the invocation. With the arrival of window systems, the advantages of object-oriented programming techniques became more pronounced, as the alternative route of interpreted languages was this time not available.

Object oriented languages support this form of code reuse through virtual methods and subtyping. The fixed top part of a program is formulated in terms of objects with virtual methods, whose implementation is omitted. Implementations of these methods are given in the bottom part of the program. An implementation is encapsulated in an object which can be substituted for an object template in the top part provided their types are in a subtype relationship.

One might argue that procedure parameters also allow reusing the top of the call graph. For instance, we could represent the command interpreter of a window with a function which has a

signature such as:

$$\textit{Handler} = \textit{State} \rightarrow \textit{Event} \rightarrow (\textit{State}, \textit{Output}).$$

The handler could be passed as a functional argument to the central dispatch routine in the window system. Parametric polymorphism [8, 16], in the form it is used in most modern functional languages, allows the formulation of a dispatch routine independently of the type of *State*, as long as the functional argument is not stored in a data structure. If handlers were collected in a list, for example, this list would have the polymorphic type

$$[s \rightarrow \textit{Event} \rightarrow (s, \textit{Output})]$$

where *s* is a type variable representing state. The crucial point is that the type variable *s* can be instantiated only once; hence, all handlers would be forced to operate on data of the same type. Since window systems commonly collect several windows with different command interpreters in a single data structure, functional arguments are insufficient.

So, if we were to program a window or simulation system in a polymorphic functional language such as ML, Miranda or Haskell, would we need object-oriented language extensions? It turns out that this is not the case. In this paper, we discuss a technique of program construction which closely models the concepts found in object-oriented programming, yet is expressible with parametric polymorphism alone; no special subtyping conventions are needed. This technique allows the reuse of the top part of the program graph in all cases where object-oriented methods allow it.

Objects, classes, self-reference and subtyping are expressed in terms of Haskell [11], a polymorphic, higher-order functional language. This gives us a transformational semantics of these concepts. Such a semantics is useful because

- the semantics of Haskell itself is well understood, it can for example be expressed as a mapping into the lambda calculus, and
- Haskell lends itself well to equational reasoning. Algebraic properties of objects and subtyping are therefore easy to prove.

The rest of this paper is organized as follows: Section 2 outlines some of the parts of Haskell which might be unfamiliar to readers used to procedural notations. Section 3 presents the functional formulation of Objects and Classes. Section 4 discusses subtyping. Section 5 discusses related work and Section 6 concludes.

2 Notation

The Haskell notation is used for the definition of functions. Functions are defined by equations. Function application is expressed by juxtaposition; the arguments of a function do not have to be

enclosed in parentheses. A Pascal function call like $f(x, y + 1)$ would be expressed in Haskell as $f\ x\ (y + 1)$. Functions can be higher order, that is, they can accept other functions as arguments or return them as results. In fact, f can be regarded as a function with a single parameter which returns another function. This convention, which is called *currying*, gives a meaning to partial function applications. For example, if *add* were defined as

$add\ x\ y = x + y,$

then

$inc = add\ 1$

would define the successor function on integers. Function application is left-associative, the expression $f\ x\ y$ is equivalent to $(f\ x)\ y$ rather than to $f\ (x\ y)$. The latter expression would be evaluated by applying f to the result of applying x to y .

Haskell is a strongly typed language which associates every legal expression with a type. The fact that an expression e has type T is expressed as $e :: T$. Types are either pre-defined like *Int* or *Char*, or they are built from other types with the following constructors:

(T_1, \dots, T_n)	Tuple with components of type T_1, \dots, T_n
$T_1 \rightarrow T_2$	Function from type T_1 to T_2
$Tag_1\ T_{1,1} \dots T_{1,k_1}$	Algebraic type denoting the disjoint sum of types consisting of
...	tag discriminator Tag_i and component types $T_{i,1} \dots T_{i,k_i}$ ($i = 1, \dots, n$).
$Tag_n\ T_{n,1} \dots T_{n,k_n}$	In this paper, we will need only algebraic types with one tag field.
$[T]$	List of type T . This is a shorthand for the algebraic type
	$List\ T = Nil \mid Cons\ T\ (List\ T)$.

The type of a function with several arguments is expressed using the currying convention. The function *add*, for instance, would be typed:

$add : Int \rightarrow Int \rightarrow Int$

The constructor \rightarrow is right-associative; $Int \rightarrow Int \rightarrow Int$ is equivalent to $Int \rightarrow (Int \rightarrow Int)$. Hence, *add 1* has type $Int \rightarrow Int$, just as required.

Types can be polymorphic, that is they can contain type variables in component position. A polymorphic type is instantiated by replacing all occurrences of a type variable with equal types. Syntactically, a type variable is distinguished from a type in that it starts with a lower case letter;

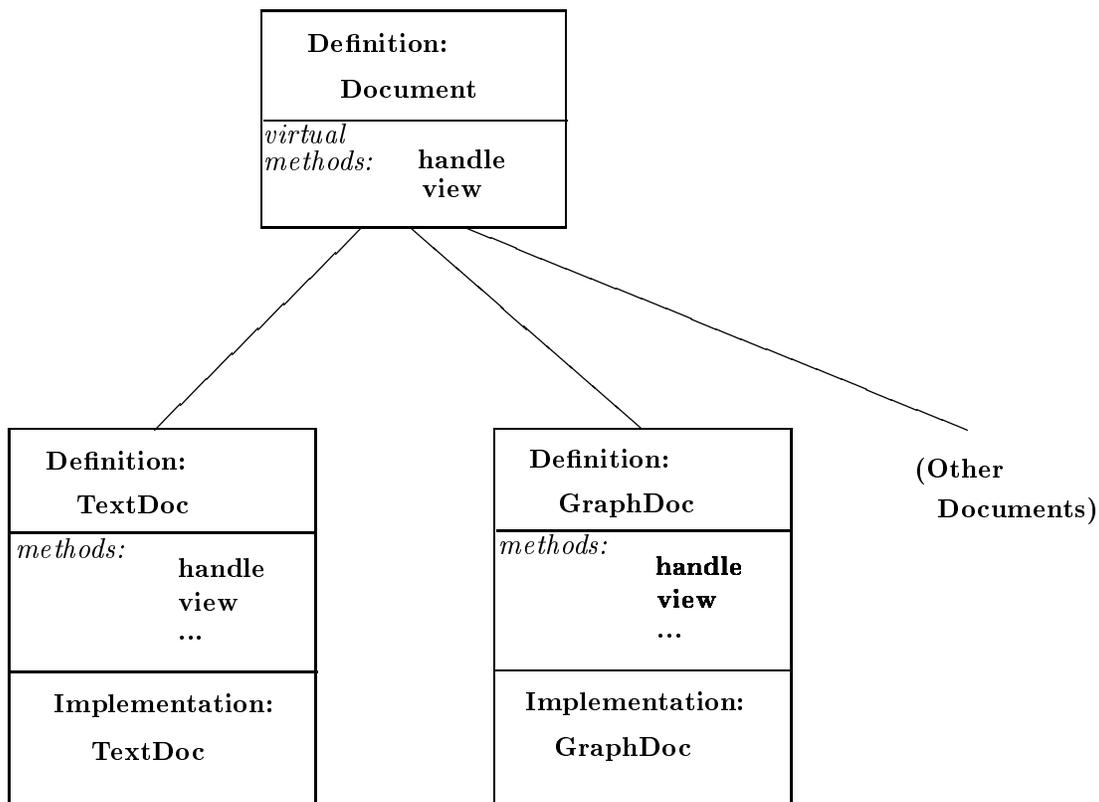


Figure 1: Example Class Hierarchy

types in contrast start with capital letters.

An important higher-order function which we will use later in the paper is the composition operator \circ , written infix. Its definition is:

$$\begin{aligned}
 (\circ) & \quad :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \\
 (f \circ g) x & = f (g x)
 \end{aligned}$$

3 Objects and Classes

Consider a window system in which every window is connected to a document. Documents can be edited and displayed. The edit-procedure has the form $handle(event)$, and the display procedure has the form $view(coords)$. The system should not impose restrictions on the type of displayed documents. For book-keeping and event-dispatch, all windows have to be collected in a central data structure such as a list or an array.

A typical object-oriented solution would involve a class hierarchy such as the one in figure 1. Different documents can be collected in one data structure because their types are all subtypes of class *Document*.

In the following, we will line out a solution to the same problem in terms of Haskell, which has no subtyping. The solution consists in translating step by step object-oriented concepts into functional ones. We start in this section with an object’s implementation and its external interface, and express the relationship between the two by means of an abstraction function. In the next section, we will explain subtyping in terms of abstraction functions from one implementation to several definitions which expose varying degree of detail.

Let us consider object implementation first. Following Wegner [21], an object consists of an internal *state* and a fixed set of *methods*, i.e. procedures and functions which access and possibly modify the object’s state. In a functional language, state modification is expressed by functions which return the modified object state as their result. We call these functions state “transformers”; they are distinguished from “observer” functions which access the state without changing it in that the result type of a transformer function equals the type of the state.

S	Object State
$ft :: S \rightarrow X \rightarrow S$	Transformer Function
$fo :: S \rightarrow Y \rightarrow Z$	Observer Function

Note: For simplicity of exposition, we model all methods as functions with two parameters. The first parameter denotes the state of the object to which the method is applied, the second denotes any additional arguments passed to it. We lose nothing in generality by doing so, because several arguments can always be collected into one by tupling them, i.e. $f\ s\ x\ y$ is transformed to $f\ s\ (x, y)$.

In most object-oriented languages, methods are declared without the first state parameter. Inside a method, the state of the “currently processed” object is denoted by the pseudo-variable *self* (or: *this*). The value of *self* can be thought of being passed to the method as an additional parameter. This is precisely what we are doing here.

Example: Consider the following class implementation of a text document:

```
class implementation TextDoc;
  t: Text;
  procedure handle(e: Event);
    .. self.t ..
  procedure view(c: Coords): View;
    .. self.t ..
  procedure repr(): String
```

```
.. self.t ..
```

It defines an object class with state of type *Text* and three methods, *handle*, *view* and *repr*. Only the first method modifies the object's internal state. This would be represented in our framework as:

```
t      :: Text
handle :: Text → Event → Text
view   :: Text → Coords → View
repr   :: Text → String

handle t e = ... t ...
view   t c = ... t ...
repr   t   = ... t ...
```

The translation models how an object is implemented and gives a meaning to the pseudo-variable *self*, but it does not adequately describe the external interface of an object. The interface differs from the implementation in that the internal state represented by instance variables is hidden. This has two effects on our functional model:

- The first “state” parameter of every method is suppressed.
- Because state is not accessible from the outside, an object's definition has to include all operations which directly access the state.

An object interface is represented as a tuple of functions. Its type has the form:

$$\begin{aligned}
 T &= TAG (sig_1^t, \dots, sig_n^t, sig_1^o, \dots, sig_m^o) & (1) \\
 sig_i^t &= X_i \rightarrow T & (i = 1, \dots, n) \\
 sig_j^o &= Y_j \rightarrow Z_j & (j = 1, \dots, m)
 \end{aligned}$$

Note: *TAG* is a constructor of an algebraic type. It is superfluous from a logical standpoint, but is required for type-checking reasons. ML style type checkers restrict type-recursion to algebraic types, a type declaration such as $T = (X \rightarrow T, \dots)$ is illegal.

The invocation of a method is performed by component selection. It can be encapsulated in another function, which shall be called a message template:

$$\begin{aligned}
 mt_i (TAG(ft_1, \dots, ft_n, fo_1, \dots, fo_m)) &= ft_i \\
 mo_j (TAG(ft_1, \dots, ft_n, fo_1, \dots, fo_m)) &= fo_j
 \end{aligned}$$

Example: The type of the *TextDoc* interface is:

$$\textit{TextDoc} = \textit{TDoc} (\textit{Event} \rightarrow \textit{TextDoc}, \textit{Coords} \rightarrow \textit{View}, \textit{String})$$

The messages applicable to an instance of *TextDoc* are defined as:

$$\textit{tdoc_handle} \quad (\textit{TDoc}(h, v, r)) = h$$

$$\textit{tdoc_view} \quad (\textit{TDoc}(h, v, r)) = v$$

$$\textit{tdoc_repr} \quad (\textit{TDoc}(h, v, r)) = r$$

How do internal implementation and external interface relate? Loosely formulated, the interface should have all properties of the implementation which do not directly reference the hidden state, and it should have only those properties. We want to represent this relationship with an abstraction function, α , from states to object interfaces. The specification of α is given by:

$$(P1) \quad \forall s \in S, i \in \{1, \dots, n\} : (mt_i \circ \alpha) s = \alpha \circ (ft_i s)$$

$$(P2) \quad \forall s \in S, j \in \{1, \dots, m\} : (mo_j \circ \alpha) s = fo_j s$$

Property P1 relates transformer messages and transformer functions. It states that the following diagram commutes:

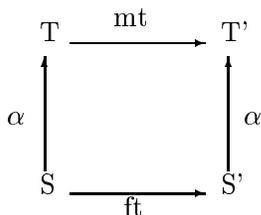


Figure 2: Property P1

Property P2 relates observer messages and observer functions. It states that applying an observer message to an abstraction gives the same result as applying the corresponding observer function to the state.

Obviously, α depends on the type constructor and the methods which define the object's behavior. We therefore define α as a function which takes a type constructor and a tuple of methods as well as a state as parameters. An implementation of α is the following:

$$\alpha \text{ tag } (ft_1, \dots, ft_n, fo_1, \dots, fo_m) s = \text{tag } (\hat{\alpha} \circ (ft_1 s), \dots, \hat{\alpha} \circ (ft_n s), fo_1 s, \dots, fo_m s)$$

where

$$\hat{\alpha} = \alpha \text{ tag } (ft_1, \dots, ft_n, fo_1, \dots, fo_m)$$

Property P1 holds, because:

$$\begin{aligned} & (mt_i \circ (\alpha \text{ tag } (ft_1, \dots, ft_n, fo_1, \dots, fo_m))) s \\ = & \{ \text{Definition of } \circ \} \\ & mt_i (\alpha \text{ tag } (ft_1, \dots, ft_n, fo_1, \dots, fo_m) s) \\ = & \{ \text{Definition of } \alpha \} \\ & mt_i (\text{tag } (\hat{\alpha} \circ (ft_1 s), \dots, \hat{\alpha} \circ (ft_n s), fo_1 s, \dots, fo_m s)) \\ = & \{ \text{Definition of } mt_i \} \\ & \hat{\alpha} \circ (ft_i s) \\ = & \{ \text{Definition of } \hat{\alpha} \} \\ & (\alpha \text{ tag } (ft_1, \dots, ft_n, fo_1, \dots, fo_m)) \circ (ft_i s) \end{aligned}$$

Property P2 holds, because:

$$\begin{aligned} & (mo_j \circ (\alpha \text{ tag } (ft_1, \dots, ft_n, fo_1, \dots, fo_m))) s \\ = & \{ \text{Definition of } \circ \text{ and } \alpha \} \\ & mo_j (\text{tag } (\hat{\alpha} \circ (ft_1 s), \dots, \hat{\alpha} \circ (ft_n s), fo_1 s, \dots, fo_m s)) \\ = & \{ \text{Definition of } mo_j \} \\ & fo_j s \end{aligned}$$

It is instructive to compare the abstraction function α with the concept of class in object oriented languages. If we leave aside the effect of a class on the visibility of the identifiers declared in it (the same effect can be achieved with a module system), the only functional aspect of a class is that it implements a procedure such as *NEW* for object creation. Hence, both abstraction functions and classes act as object generators. A difference between them lies mainly in parameterization: Unlike α , *NEW* does not take method implementations as parameters. The same effect can be achieved in our framework by instantiating α with a fixed method tuple. For instance, the object generator of class *TextDoc* could be formulated as:

$$\text{textdoc } t = TDoc (\text{textdoc} \circ \text{handle } t, \text{view } t, \text{repr } t)$$

This is possible as long as the implementation of a method is known in the context of the class, or, put in other words, as long as methods are not virtual. A virtual method appears as a functional

argument to the object generator.

Example: The class *Document* with its two virtual methods, *handle* and *view*, would be identified with the object generator

$$\begin{aligned} \textit{Document} &= \textit{Doc} (\textit{Event} \rightarrow \textit{Doc}, \textit{Coords} \rightarrow \textit{View}) \\ \textit{doc} (\textit{handle}, \textit{view}) s &= \textit{Doc} (\textit{doc} (\textit{handle}, \textit{view}) \circ \textit{handle} s, \textit{view} s) \end{aligned}$$

4 Subtyping

One important aspect of the abstraction function α is that it is polymorphic in the type of the state parameter. Hence, different implementations can be abstracted to the same object interface. This makes it possible to construct heterogeneous data structures.

Example: The abstraction function *doc* is typed

$$\textit{doc} :: (s \rightarrow \textit{Event} \rightarrow s, s \rightarrow \textit{Coords} \rightarrow \textit{View}) \rightarrow s \rightarrow \textit{Doc}$$

An object implementation can be abstracted to a document provided it has two methods f_1, f_2 with signatures

$$\begin{aligned} f_1 &:: T \rightarrow \textit{Event} \rightarrow T \\ f_2 &:: T \rightarrow \textit{Coords} \rightarrow T \end{aligned}$$

There are no restrictions on type T . One implementation which obviously meets the requirements for document abstraction is the implementation of a text document, but others are just as well possible. We could think of a graphic document, whose internal state is a graph rather than a text, and which implements the functions:

$$\begin{aligned} \textit{handle} &:: \textit{Graph} \rightarrow \textit{Event} \rightarrow \textit{Graph} \\ \textit{view} &:: \textit{Graph} \rightarrow \textit{Coords} \rightarrow \textit{View} \end{aligned}$$

By abstracting text and graphic documents to the same type, they can be used interchangeably as components of other types, for instance as component of:

$$\textit{Window} = \textit{Win} \dots \textit{Doc} \dots$$

We have hence arrived at a solution for the problem we started with, namely how to reuse the

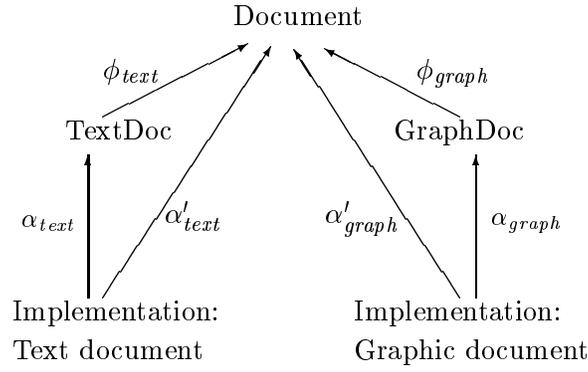


Figure 3: Abstraction and Coercion Functions

top part of the call graph. However, the analogy to object-oriented subtyping is not very strong. Abstraction functions provide a mapping from implementations to interfaces while true subtyping would involve coercions between interfaces. Looking at figure 3, we have defined abstraction functions α'_{text} and α'_{graph} . Subtyping would allow us to use a *TextDoc* or *GraphDoc* in place of a *Document* by implicitly applying the coercion functions ϕ_{text} or ϕ_{graph} .

True subtyping can be implemented by using the fact that abstraction functions can be abstracted themselves. The technique shall be illustrated with the *TextDoc* example at hand:

```

TextDoc = TDoc (Event → TextDoc,
                Coords → View,
                String,
                Document)
  
```

```

tdoc_doc:: TextDoc → Document
tdoc_doc (TDoc h v r d) = d
  
```

```

textdoc:: Text → TextDoc
textdoc t = TDoc (textdoc ∘ handle t, view t, repr t, doc (handle, view) t)
  
```

The above code fragment shows the extended interface of a text document, an additional template for a message *tdoc_doc*, which performs the coercion from *TextDoc* to *Document*, and the modified object generator *textdoc*.

What we have done is to define a coercion ϕ between object interfaces T and T' which share the same implementation I in terms of the two abstraction functions from I to T and T' , respectively. Formally, we have:

$$\phi \circ \alpha = \alpha'$$

The question now arises whether a definition of ϕ without reference to α and α' can be given. The question can be answered in the affirmative if the implementations of the common methods of T and T' are the same. Let T be given as in (1), and let T' be defined as

$$T' = TAG' (ft_1, \dots, ft_{n'}, fo_1, \dots, fo_{m'}) \quad \text{where } n' \leq n, m' \leq m$$

Then the coercion function ϕ from T to T' is given by:

$$\phi TAG (ft_1, \dots, ft_n, fo_1, \dots, fo_m) = TAG' (\phi \circ ft_1, \dots, \phi \circ ft_n, fo_1, \dots, fo_m')$$

Proof: For fixed $TAG, ft_1, \dots, ft_n, fo_1, \dots, fo_m$, define:

$$\begin{aligned} \hat{\alpha} &= \alpha TAG (ft_1, \dots, ft_n, fo_1, \dots, fo_m) \\ \hat{\alpha}' &= \alpha' TAG' (ft_1, \dots, ft_{n'}, fo_1, \dots, fo_{m'}) \end{aligned}$$

We have:

$$\begin{aligned} &(\phi \circ \hat{\alpha}) s \\ &= \phi (TAG (\hat{\alpha} \circ ft_1, \dots, \hat{\alpha} \circ ft_n, fo_1, \dots, fo_m)) \\ &= TAG' (\phi \circ \hat{\alpha} \circ ft_1, \dots, \phi \circ \hat{\alpha} \circ ft_{n'}, fo_1, \dots, fo_{m'}) \end{aligned}$$

The above equations simply substitute the definitions of $\hat{\alpha}$, ϕ , and \circ in the initial expression $\phi \circ \hat{\alpha}$. This expression therefore equals the least fixpoint of the function

$$\lambda f. TAG (f \circ ft_1, \dots, f \circ ft_{n'}, fo_1, \dots, fo_{m'})$$

But this is also the definition of $\hat{\alpha}'$! Because the least fixpoint of a lambda expression is unique it follows that

$$\phi \circ \hat{\alpha} = \hat{\alpha}'.$$

Since ϕ does not make use of abstraction functions, it can be defined after the code of class generators is fixed. No order on the definition of classes is imposed. It therefore implements class generalization as discussed in [17].

We have assumed in the preceding discussion that the method implementations shared by T and T' are the same. This need not always be the case, however. The method implementations

passed to the abstraction functions of a class and its superclass may well differ. In particular, methods in the supertype can be derived from those in the subtype by composition with coercion operators. A method $f' :: X' \rightarrow Y'$ in T' corresponding to a method $f :: X \rightarrow Y$ in T may take the form

$$f' = \phi_{Y, Y'} \circ f \circ \phi_{X', X}$$

where $\phi_{A, B}$ denotes the coercion function from A to B . This amounts to adopting the standard “contra-variant” subtyping rule for functions [3]:

$$T_1 \rightarrow T_2 \leq T'_1 \rightarrow T'_2 \quad \text{if} \quad T'_1 \sqsubseteq T_1 \wedge T_2 \sqsubseteq T'_2$$

5 Related Work

The emulation of object-oriented concepts in terms of notations which do not offer them directly is by no means new. However, all work in this area which we know of either relies on macro-expansion and recompilation of reused code, or breaches the type-system by making use of unsafe features such as memory addresses. A technique which is close to ours is presented in [1]. Abelson and Sussman emulate objects in the Scheme notation by functions which hide state in a local environment. These functions return an association list of labels and local procedures. Method dispatch is implemented by selecting the local procedure corresponding to a given label. Since Scheme is not statically typed, the notion of subtyping is not meaningful and is hence left out. We do not know of previous work which discusses this style of object abstraction in a strongly typed framework.

Abstraction functions from type implementations to type definitions were first introduced in [9] and were further developed in algebraic data type theory [7, 20]. These works concentrate on correctness proofs of a data type’s implementation with respect to its specification; the case where the interface (i.e specification) and the implementation are parts of the same program is not considered. Programming languages such as CLU [13] which incorporate abstract data types establish the relation between interface and implementation by a reverse mapping, *rep*, from the abstract type to the concrete one, thereby preventing the possibility of having more than one implementation for a given interface.

There are several works which touch the relationship between parametric polymorphism and subtyping. Meyer [15] argues that subtyping is not subsumed by parametric polymorphism (which he calls genericity). His counter-example involves assignment statements and is similar in spirit to the problem from which we started — how to construct data structures of components whose precise type is not known. We believe to have disproved that claim in our paper. However, our technique rests on the use of higher-order functions, a concept which Meyer did not consider in his work.

The language Haskell introduced “type-classes” which very much resemble in notation common object-oriented concepts. As is shown in [19] and [12], type classes can be expressed with just parametric polymorphism. Despite the similar notation, there is an important difference to subtyping: A type class describes properties of a type, not the type itself. The hierarchy of properties does not extend to a subtype relationship between types. Type classes allow the formulation of functions which operate uniformly on all type instances of descendants of a given class, but they do not permit the construction of heterogeneous data structures. Similar in spirit are the descriptive classes in [18], and F-bounded polymorphism [2]. The latter combines the concepts of property- and type- inheritance in a single framework.

Our transformational semantics for objects and subtyping complements several other works which develop a denotational semantics for these concepts. Cardelli [3] presents such a semantics for multiple subtyping in a type system comprising record types and variants; object implementation is not discussed. Cook [4] expresses objects as least fixpoints of functions from “self” to a record type. This opens up the possibility of modifying the generating function before forming the fixpoint and thus provides a semantics for method inheritance as opposed to subtyping [5]. Method inheritance is not supported in our framework.

6 Conclusions

The technique presented in this paper emulates the concepts of object interface and implementation, class, and subtyping in a strongly typed, polymorphic language. The key for the implementation of subtyping is the suppression of a function’s first state parameter in the interface. If the first parameter were not suppressed, it would be submitted to the contra-variant rule, and this is not what we want.

By making coercions between objects explicit, we get a transformational semantics of subtyping. A function application $f x$ where $f :: T_1 \rightarrow T_2$, $x :: T$, $T \sqsubseteq T_1$ is equivalent to $f (\phi_{T, T_1} x)$. The latter expression is typeable in a type system with just parametric polymorphism.

There are also consequences for functional programming. As we suggested in the introduction, subtyping is needed for re-using the top part of a call graph. In some important problem areas such a style of code reuse is indicated; we have mentioned simulation and window systems. Our emulation of subtyping allows languages with Hindley/Milner style polymorphism to be used in these areas.

This does not imply that object-oriented languages with subtype-polymorphism are unnecessary. Some points where our technique differs from object-oriented languages are:

- Coercion functions have to be written explicitly.
- Method implementations in a superclass can not automatically be reused in subclasses. Reuse of such methods can in most cases be achieved by exporting them from the scope of the

superclass and using them in the abstraction of the subclass. This requires explicit program code.

- Classes are just generator functions, they do not open a new scope for their methods and instance variables. Scope rules to that effect have to be provided by a module system.
- Explicit subtyping as we presented it carries a performance penalty. Every evaluation of a transformer function requires the construction of a new method tuple. This overhead is to some extent inherent in functional languages, which do not allow destructive updating of just the object state. A possible solution to this problem is presented by single-threaded type systems [10], which support destructive updating in a referentially transparent way.

One area which we have not covered so far is multiple inheritance. While multiple inheritance is supported by our technique (abstraction functions can be constructed for more than one superclass of an implementation), it is not as useful as in the context of procedural programming languages. Remember that the purpose of emulating subtyping was to combine different implementations in a common data structure. The main benefit of multiple inheritance seems to be that one object can appear in several data structures of different type. But this makes sense only as long as the object is not going to be changed. Sending a transformer message to an object will construct a new object. Explicit code is required to substitute the transformed object for the original one in all structures which refer to the original. This might be impossible, because the structures referring to an object are not necessarily known at the point where the transformer message is sent.

Updating an object in a procedural language, on the other hand, changes as a side effect all data structures referring to the object. Such side effects are criticized for not having a simple semantics, but they seem to be often employed in object-oriented programs. More practical experience is needed to answer the question whether the judicious use of side effects is needed for the type of problems solved by object-oriented programming, or whether a referentially transparent programming style would do just as well or even better.

References

- [1] Abelson H., Sussman G.J., with Sussman J., Structure and interpretation of computer programs. MIT Press, 1985.
- [2] Canning P., Cook W., Hill W., Olthoff W., and Mitchell J.C., F-bounded polymorphism for object-oriented programming. In Proc. Conf. on Functional Programming Languages and Computer Architecture, pages 273–280, 1989.
- [3] Cardelli L., A semantics of multiple inheritance. In Semantics of Data Types, LNCS 173, pages 51–67, 1984.
- [4] Cook W.R. and Palsberg J., A denotational semantics of inheritance and its correctness. In Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications, pages 433–443, 1989.
- [5] Cook W.R., Hill W.L., and Canning P.S., Inheritance is not subtyping. In Proc. Conf. on Principles of Programming Languages, pages 125–135, 1990.
- [6] Dahl O.-J., Nygaard K., Simula: An Algol-based simulation language. *Comm. ACM* 9(9), pages 671–678, 1966.
- [7] Guttag J.V., Abstract data types and the development of data structures. *Comm. ACM* 20(6), pages 396–404, 1977.
- [8] Hindley R., The principal type scheme of an object in combinatory logic. *Trans. Am. Math. Soc.* 146, pages 29–60, 1969.
- [9] Hoare C.A.R., Proof of correctness of data representations. *Acta Informatica* 1(4), pages 271–281, 1972.
- [10] Hudak P. and Guzman J.C., Taming side effects with a single-threaded type system. Draft.
- [11] Hudak P. and Wadler P. (editors), Report on the functional programming language Haskell, Technical Report YALEU/DCS/RR666, Yale University, 1988.
- [12] Kaes S., Parametric overloading in polymorphic programming languages, In Proc. European Symposium on Programming, pages 131–144, LNCS 300, 1988.
- [13] Liskov B. et al., CLU Reference Manual, LNCS 114, 1981.
- [14] Liskov B., Data abstraction and hierarchy. In Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications, pages 17–34, 1987.
- [15] Meyer B., Genericity versus Inheritance. In Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications, pages 391–405, 1986.

- [16] Milner R.A., A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* 17(3), pages 348–375, 1978.
- [17] Pedersen C.H., Extending ordinary inheritance schemes to include generalization. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 407–417, 1989.
- [18] Sandberg D., An alternative to subclassing. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 424–428, 1986.
- [19] Wadler P., How to make ad-hoc polymorphism less ad hoc. In *Proc. Conf. on Principles of Programming Languages*, pages 60–76, 1989.
- [20] Wand M., Final algebra semantics and data type extensions. *J. Comput. Syst. Sci.* 19(1), pages 27–44, 1979.
- [21] Wegner P., Dimensions of object-based language design. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 168–182, 1987.