ISSN1000-9825 Journal of Software 软件学报 2000,11(1):8~22

Combining OO and Functional Language Concepts*

QIAN Zhen-yu Besma Abd Moulah

(Universität Bremen Germany)

E-mail: qian@kestrel.edu

Abstract This paper considers the problem of combining the object-oriented and functional programming paradigms. Compared with most of the approaches in this direction, the combination has the following two advantages. First, the authors combine several important concepts as they are well known in widespread mainstream languages. In other words, the authors do not introduce new language concepts but try to interpret well-known language concepts based on the new ones. Second, the combination has the property that individual language concepts do not influence the whole language to the extent as they do traditionally, so that usually one needs to pay for a language concept only when he uses it. Concretely, a core language for functional object-oriented programming together with a straightforward operational semantics is proposed, where the properties mentioned above hold. The core language combines the following key language concepts from the languages Eiffel, Java, ML and Haskell: objects, classes, multiple inheritance, method redefinition, dynamic binding, static type safety, binary methods, algebraic data types, higher-order functions, ML-polymorphism.

Key words Object-Oriented programming, multiple inheritance, method redefinition, dynamic binding, static type safety, binary methods, functional programming, algebraic data types, higher-order functions, ML-polymorphism.

A programming language should provide concepts for coding (real-world) objects such as persons and cars, and algebraic values such as integers, reals and lists. Object-Oriented (programming) languages can naturally describe objects, while functional (programming) languages can naturally construct algebraic values by function symbols. Thus it is desirable to combine both paradigms.

Several languages, e.g. Pizza[1], Objective ML[2], Object SML[3,4], Haskell[5], were quite successful in combining different ideas from both paradigms. Whether these combinations will reach widespread industrial acceptance is still open. In our view, a problem with these languages might be that some important concepts in widespread mainstream languages are missing. Another problem might be that these languages often embody well-known ideas by language concepts that are different from those in widespread mainstream languages. Indeed, new language concepts can be desirable and necessary in many situations. But, if possible, a better approach might be to combine the key language concepts from mainstream languages, since these concepts have already proved successful in many different applications.

Since the existing language concepts do not always mix well, we pursue a discipline in our combination,

* **QIAN Zhen-yu** is currently a computer scientist at Kestrel Institute. Prior to joining Kestrel, he held academic positions at Bremen University in Germany, where he received Ph. D. and "Habilitation" in Computer Science in 1991 and 1996, respectively. His research interests include object-oriented, functional, concurrent, logic programming and specification languages. **Besma Abd Moulah** is a Ph. D. candidate in Computer Science. Her interests include programming and specification languages, and formal methods for software engineering.

Manuscript received 1998-05-06, accepted 1998-10-30.

namely "you pay only for the concepts you use". The discipline is important for at least two reasons: First, programmers usually expect that the same concepts have comparable behavior and performance; second, it should be possible to implement only a part of these concepts without losing much in efficiency for concepts that are not implemented.

This paper describes, as the first steps of a long-term attempt, how to combine key language concepts from the languages Eiffel, Java, ML and Haskell. Haskell is particularly important, since its type class system provides a natural connection between the object-oriented and functional paradigms, i.e. between interfaces and algebraic data types. We propose a core language FOC* for functional object-oriented programming and define an operational semantics in a simple and straightforward way. The concepts in consideration are objects, classes, multiple inheritance, method redefinition, dynamic binding, static type safety, binary methods, algebraic data types, higher-order functions, ML-polymorphism.

At least two important issues are not considered in FOC. The first is the design of a type reconstruction. The reason is that the problem can be made orthogonal to FOC by assuming that a type reconstruction has been run and all type annotations have been generated. Since too many explicit type annotations may make a program look involved, we write only explicit casting operators where subtyping is needed and give a usual type inference system to infer other type annotations. Note that a powerful type reconstruction can be in general a difficult task in the presence of subtyping and type variables: a type reconstruction may easily yield a huge amount of subtyping constraints[6], which can neither be easily checked nor be simplified (see Refs. [7~9]).

Another important issue we do not consider is the concept of references. Although the related language concepts, e.g. object states, the assignment, the value or reference semantics of an expression, classes and inheritance, and interfaces and implementation, are important in object-oriented programs, and although we use the assignments in sample programs in this paper, the issue is to a great extent orthogonal to the topics in consideration and will be documented in a separate paper.

1 Syntax of FOC

The syntax of FOC is given in Fig. 1, where α,β range over *type variables*, x over *expression variables*, κ ranges over *classes*, χ over *type constructors* (of algebraic data types), o_κ over objects of class κ, a over ***attributes*** and c over *value constructors* (of algebraic data types), m over *methods*, and $n,h\geq 0$. We assume the usual convention for renaming bound variables. Throughout the paper, the notation $\overline{O_n}$ denotes the list $O_1,\ldots,O_n$ with $n\geq 0$.

$$
\begin{array}{lll}
\text{Monomorphic types} & \tau & = \alpha \mid \kappa \mid \chi\ (\overline{\tau_n})\ (\text{for } n\text{-ary } \chi) \mid (\overline{\tau_n}) \mid \tau_1 \rightarrow \tau_2 \\
\text{Polymorphic types} & \sigma & = \tau \mid \forall\ \alpha \prec \kappa.\ \sigma \\
\text{Expressions} & s,t & = x \mid m \mid c(\overline{s_n}) \mid (\overline{s_n}) \\
 & & \quad \mid \lambda x:\tau.\ s \mid (s\ t) \mid \Lambda\alpha \prec \kappa.\ s \mid s[\tau] \\
 & & \quad \mid \textbf{let } x:\sigma = s \textbf{ in } t \\
 & & \quad \mid o_\kappa \mid s.\,a \mid s \uparrow \tau \\
\text{Values} & u,v & = m\ [\overline{\tau_n}] \mid c(\overline{s_n}) \mid (\overline{s_n}) \mid \lambda x:\tau.\ s \mid o_\kappa \mid s.\,a \\
\text{Declarations} & d & = \textbf{class } \kappa \prec K \textbf{ attr } \boldsymbol{a}:\xi \textbf{ meth } m\lfloor :\sigma \rfloor = s \\
 & & \quad \mid \textbf{datatype } \chi(\overline{\alpha_n}) \textbf{ cons } c:\sigma \\
 & & \quad \mid \textbf{subtype } \overline{\alpha_k \prec \kappa_h} \Rightarrow \chi(\overline{\alpha_h}) \prec \kappa \textbf{ meth } m = s \\
 & & \quad \mid \textbf{fun } x:\sigma = s \\
\text{Programs} & p & = s \mid d;\ p
\end{array}
$$

Fig. 1 Syntax of FOC

* The name stands for "Functional Object-Oriented language Concepts".

1.1 Types

Classes in FOC combine object classes in Eiffel and type classes in Haskell. Type constructors build algebraic data types.

Monomorphic types are constructed by type variables, classes and *algebraic data types*. *Function types* $\tau_1 \rightarrow \tau_2$ and *n-tuple types* $(\overline{\tau_n})$ are algebraic data types constructed by predefined type constructors $\rightarrow$ and $(\text{-},\ldots,\text{-})$. Monomorphic types containing no type variables are called *ground* types. Ground types are ranged over by ξ.

Polymorphic types are obtained from monomorphic types by all-quantifying type variables at the outermost position using the subtyping relation $\prec$ and classes as *bounds*. The definition of $\prec$ will be given later. A polymorphic type is said to be *closed* if it contains no free type variables.

1.2 Class declarations

A class declaration (cf. the syntax in Fig. 1) declares a *class* κ, which is a *subclass* of each class, called ***superclass***, in K, ***declares*** an ***attribute*** by an *attribute declaration* $a:\xi$, and *declares* a method m by a *method declaration* $m:\sigma=s$ or ***redefines*** the method m by a *method redefinition* $m=s$. The syntax supports the multiple inheritance, since K may contain more than one class. In general, a class may contain zero or more attribute and method declarations. We write the syntax with one piece of each for notational simplicity. A class cannot be declared more than once in a program.

An attribute a can only be of a ground type ξ. This is a consequence of several design decisions. First, classes in FOC are at the top-level and thus the type of an attribute should be closed. Furthermore, we decide that the type of an attribute in FOC should not have any quantifications of type variables; otherwise a more complex type system for deep polymorphism would be needed[10~12].

An attribute of a class is either *declared* in the class or *inherited* from a superclass. For simplicity, we require that if an attribute has been declared in a superclass then it cannot be declared again in a subclass.

For simplicity, we also require that a method can be declared at most once in a program. If a method is declared, then it can be redefined in subclasses.

The requirement that a method can be declared at most once in a program may be too restrictive for practical programming. We do this just for notational simplicity. In fact, one could relax the restriction in several ways. The details are out of the scope here.

A method declaration $m:\sigma=s$ consists of a method name m, a closed polymorphic type σ and a *body* s. The name m may be used in the body s or the bodies of other methods. Thus methods may be mutually recursive. Although the type of a method is a polymorphic type, the type inference system (in Section A.4) will ensure that a method application is always of a monomorphic type.

The type σ of a method declaration $m:\sigma=s$ in the class κ must be closed and of the form

$$\forall\ \alpha\prec\kappa.\ \forall\ \overline{\alpha_n\prec\kappa_n}.\ \underbrace{(\alpha,\ldots,\alpha)}_{g}\rightarrow\tau$$

for $g\geq 1$ and $n\geq 0$. The type variable α of the outermost quantification resembles the type of self-object in a usual object-oriented language and thus always has the current class κ as its bound. The part $(\alpha,\ldots,\alpha)$ is the type of the first argument. One may regard the first argument as a receiver-expression of a method. (One can write e.g. $(x,y).eq$ instead of $eq(x,y)$.) In order to ensure static type-safety, we require that the type variable α should never occur within the domain type of a function type in the type τ. If $g=1$, then the method is called *simple* and the type (α) of the first argument may be written as α; otherwise, the method is called *binary*.

If a class κ' declares a method by $m:\sigma'=s$ and a class κ is a subclass of κ', then κ can redefine the method m by $m=t$.

A class κ is said to inherit a method m from a superclass κ' if κ' declares or redefines the method m and there are no subclasses of κ' that are superclasses of κ and declare or redefine the method m.

A method of a class is either declared or redefined in the class or inherited from a superclass.

In the presence of method redefinitions we need dynamic binding to determine a method body for a method application (Section 3).

Figure 2 contains three classes in FOC. The class *Eq* consists of only a binary method *eq* for testing the equality of two objects. The class *Point* is a subclass of *Eq*. It declares an attribute *pos* for positions and a simple method *move* for moving an object to a position, and redefines the method *eq*. The class *ColorPoint* is a subclass of *Point*, declares an additional attribute *color*, inherits the attribute *pos* and the method *move* from *Point* and redefines the method *eq*.

```
class Eq
  meth eq: ∀ α<Eq. (α,α)→Bool
       eq (x,y)=eqObj(x,y)
class Point<Eq
  attr pos:Int
  meth move: ∀ α<Point. α→Int→α
       move x i=(x. pos:=i)
  meth eq(x,y)=eq(x. pos,y. pos)
class ColorPoint<Point
  attr color:Int
  meth eq(x,y)=eq(x. pos,y. pos) & eq(x. color,y. color)
```

Fig. 2 Classes in FOC

As in most functional languages, methods in the program are defined using *patterns*[*]. Explicit type parameters are omitted for simplicity. It is clear that the methods can be transformed into the FOC syntax in a standard way. We assume that the type *Bool* has the usual values and the usual operation &, that the type *Int* has the usual values and a method *eq* (see Section 1.4), and that an operation *eqObj* for checking the equality of two objects is predefined.

1.3 Declarations of algebraic data types

A datatype declaration (cf. the syntax in Fig. 1) declares a type constructor χ with a value constructor c of the type σ. The type σ must be closed and of the form $\forall\, \overline{\alpha_n}.\, (\overline{\tau_h})\rightarrow\chi(\overline{\alpha_n})$. A type constructor can be declared at most once in a program. A value of an algebraic data type $\chi(\overline{\tau_h})$ must be of the form $c(\overline{t_h})$ where each t_i is an expression of the type τ_i. In general we may write zero or more value constructors in one datatype declaration. The syntax here includes only one value constructor for notational simplicity. As an example, the list type constructor *List* can be declared as in Fig. 3. The predefined types *Int* and *Bool* are regarded as algebraic data types.

```
datatype List(α)
  cons empty: ∀ α. List(α)
  cons con: ∀ α. (α,List(α))→List(α)
```

Fig. 3 A datatype declaration in FOC

1.4 Subtype declarations

A subtype declaration (cf. the syntax in Fig. 1) declares that a type $\chi(\overline{\tau_h})$ is a subtype of class κ whenever each τ_i is a subtype[**] of κ_i[***], and redefines a method of κ by $m=s$. A type $\chi(\overline{\tau_h})$ as above *inherits* each method of κ that is not redefined in the subtype declaration.

In general, a subtype declaration may contain zero or more method redefinitions. The syntax includes only one redefinition for notational simplicity.

The style of subtype declarations resembles that of instance declarations in Haskell. The intuition behind the subtype declaration is that the class κ is an interface and each type $\chi(\tau_h)$ implements the interface. But we

[*] Patterns are constructed by value constructors and expression variables in the standard way. In particular, a pattern does not contain casting operators.

[**] The subtyping relation is defined in Section A.3.

[***] Although we could have allowed a set of classes at the place of κ_i, we did not do it for simplicity.

want to keep the simplicity and do not explicitly introduce a concept of interface. Thus we require that the class κ in the above subtype declaration should contain no attributes. Since a subtype declaration does not declare a new type, we require that a subtype declaration cannot declare attributes or new methods.

```
subtype $Int < Eq$
  meth $eq(x,y) = eqInt(x,y)$
subtype $\alpha < Eq \Rightarrow List(\alpha) < Eq$
  meth $eq(empty, empty) = True$
       $eq(empty, con(y,ys)) = False$
       $eq(con(x,xs), empty) = False$
       $eq(con(x,xs), con(y,ys)) = eq(x,y)\ \&\ eq(xs,ys)$
```

Fig. 4　Subtype declarations

We can extend the declarations in Figs. 2 and 3 by the subtype declarations in Fig. 4. The first subtype declaration declares that *Int* is a subtype of *Eq*, and redefines the method *eq* in *Int* using a predefined operation *eqInt*. The second subtype declaration declares that a list type is a subtype of *Eq* if the element type is a subtype of *Eq*, and redefines the method *eq* for these list types. Note that the body of the redefinition for the list types contains applications of the method *eq* on list elements. These applications are statically well-typed, since the type of the elements is a subtype of *Eq*.

1.5　Function declarations

Function declarations define usual functions as in functional languages. The functions are global and can be used in class and subtype declarations.

1.6　Expressions

The expressions are divided into three groups. The first group consists of expression variables x, method names m, algebraic expressions $c(\overline{s_n})$, n-tuple expressions $(\overline{s_n})$, function abstractions $\lambda x:\tau.s$, function applications $(s\ t)$, type abstractions $\Lambda\alpha<\kappa.s$, type applications $s[\tau]$ and let-constructs **let** $x:\sigma=s$ **in** t. All the above expressions are Core-XML like expressions[13]. We regard n-tuple expression $(\overline{s_n})$ as being constructed by a predefined n-tuple value constructor $(-,\ldots,-)$. We may write $(s\,t_1\ldots t_n)$ for $(((s\,t_1)\ldots)t_n)$ and $s[\tau_1,\ldots,\tau_n]$ for $s[\tau_1]\ldots[\tau_n]$. Note that the definition $x=s$ in **let** $x:\sigma=s$ **in** t is assumed to be not recursive for the sake of simplicity. Although the variable x in **let** $x:\sigma=s$ in t is annotated explicitly with a polymorphic type σ, the type inference system (in Section A.4) will ensure that each occurrence of it in t is of a monomorphic type.

The second group of expressions in FOC is those for creating and manipulating objects. Here we consider only a notation o_κ standing for an object of the class κ and a notation $s.a$ for the usual attribute access operation on an object. As mentioned, we do not consider references in this paper. If we did this, then we would have to consider a notion of the object attribute assignment $s.a:=t$, which assigns t to the attribute a of the object s, a sequence expression $(s_1;\ldots;s_n)$, which executes the expressions $s_1,\ldots,s_n$ in that order, an environment to map each object attribute to a reference, and an environment to map each reference to an expression. A consequence would be also that we would have to decide which expression should have the value semantics and which should have the reference semantics. Since the language allows no implicit type coercion (see below) and the static type inference assigns a unique static type to each expression, we could do the following: If the type is a class then the expression would have the reference semantics; if the type is an algebraic data type then the expression would have the value semantics; if the type is a type variable then the expression must be a bound variable of an enclosing abstraction. A bound variable of an abstraction need not have the reference nor value semantics in our language, since it can be implemented as a placeholder, whose semantics is that of the actual argument. Although these language concepts must exist in object-oriented programs, the treatment is greatly orthogonal to the topic here.

The third group of expressions is of the form $s\uparrow\tau$, called a *casting conversion*. The notation $\uparrow\tau$ is called a *casting operator*, which converts an expression of a subtype into one with the supertype τ.

1.7 Values

In this paper values are only used to formulate the dynamic binding (in Section 3). Thus they need not be completely irreducible. More concretely, a value may be a method name, an algebraic expression, a tuple, an abstraction, an object creation or an attribute access. In particular, a value does not have casting operators at the outermost position.

After the declarations in Figs. 2, 3 and 4, we can write the following expressions:

$$\textbf{let } x = con(o_{\mathrm{ColorPoint}} \uparrow Point, con(o_{\mathrm{Point}}, empty)) \textbf{ in}$$
$$\textbf{let } y = con(o'_{\mathrm{ColorPoint}} \uparrow Point, con(o'_{\mathrm{Point}}, empty)) \textbf{ in } eq(x, y)$$

which compare two lists via the method *eq*. The lists are statically well-typed.

1.8 The environment

This subsection defines several environment components for a given program.

Ω is the set of the formulas $\kappa \prec \kappa'$ or $\forall\, \overline{\alpha_h \prec \kappa_h} \Rightarrow \chi(\overline{\alpha_h}) \prec \kappa$ in all class or subtype declarations.

Σ is the set of the pairs (m, σ) for all method declarations $m : \sigma = s$. Note that the class κ is also included in the method type σ; we prefer to have it explicitly for convenience. The method type in the method declaration always generalizes that in a redefinition. Thus Σ need not include the type in a redefinition.

Γ denotes a sequence of formulas of the form $\alpha \prec \kappa$ or $x : \sigma$, where $\alpha \prec \kappa$ *declares* a type variable α with a class κ as *bound*, and $x : \sigma$ an expression variable x of a polymorphic type σ. A sequence Γ may be written as $[\ldots]$ and $+$ denotes the concatenation operation.

An environment contains Ω, Σ and Γ. Since Ω and Σ are fixed in a given program, we will omit them in the inference rules.

1.9 Additional restrictions on the syntax

The class and subtype declarations in a program induce a subtyping relation on types. A formal definition of the relation is given in Section A. 3. In this section we will use the special case $[\,] \vdash \xi \prec \xi'$, which is a reflexive and transitive relation satisfying the following conditions:

- $[\,] \vdash \kappa \prec \kappa'$ holds for each $\kappa \prec K \in \Omega$ and $\kappa' \in K$.
- For each $\overline{\alpha_h \prec \kappa_h} \Rightarrow \chi(\overline{\alpha_h}) \prec \kappa \in \Omega$, if $[\,] \vdash \xi_i \prec \kappa_i$ hold for $i = 1, \ldots, h$, then $[\,] \vdash \chi(\overline{\xi_h}) \prec \kappa$ holds.

Two classes κ and κ' are said to be *independent* if none of $[\,] \vdash \kappa \prec \kappa'$ and $[\,] \vdash \kappa' \prec \kappa$ holds.

Now we formulate the following additional restrictions on the declarations of a program.

1. There is a predefined class Obj such that $[\,] \vdash \kappa \prec Obj$ holds for each κ, and $\forall\, \overline{\alpha_h \prec Obj} \Rightarrow \chi(\overline{\alpha_h}) \prec Obj \in \Omega$ holds for each χ.
2. There is at most one $\forall\, \overline{\alpha_h \prec \kappa_h} \Rightarrow \chi(\overline{\alpha_h}) \prec \kappa \in \Omega$ for each χ and κ. For each $\forall\, \overline{\alpha_h \prec \kappa_h} \Rightarrow \chi(\overline{\alpha_h}) \prec \kappa \in \Omega$, and each κ' with $[\,] \vdash \kappa \prec \kappa'$, there is always $\forall\, \overline{\alpha_h \prec \kappa'_h} \Rightarrow \chi(\overline{\alpha_h}) \prec \kappa' \in \Omega$ such that $[\,] \vdash \kappa_i \prec \kappa'_i$ hold for $i = 1, \ldots, h$.
3. For any $(m, \forall\, \alpha \prec \kappa' . \ldots) \in \Sigma$, the following conditions hold:
   - All subtype declarations $\forall\, \overline{\alpha_h \prec \kappa_h} \Rightarrow \chi(\overline{\alpha_h}) \prec \kappa$ with $[\,] \vdash \kappa \prec \kappa'$ redefine m with the bodies $\Lambda\, \overline{\alpha_h \prec \kappa_h}. \Lambda\, \overline{\alpha'_k \prec \kappa'_k}. t$ containing identical $\Lambda\, \alpha'_n \overline{\prec \kappa'_n}. t$, or none of these subtype declarations redefine m.
   - For each $\forall\, \overline{\alpha_h \prec \kappa_h} \Rightarrow \chi(\overline{\alpha_h}) \prec \kappa \in \Omega$ with $\kappa \not\equiv \kappa'$ and $[\,] \vdash \kappa \prec \kappa'$, the class κ contains no redefinition of m.
4. For any $(m, \forall\, \alpha \prec \kappa' \ldots) \in \Sigma$, if the classes κ_1 and κ_2 are both subclasses of κ', and have a common subclass κ, then one of the following conditions is true:
   - Both κ_1 and κ_2 inherit the method m from a common superclass.
   - Every maximal common subclass of κ_1 and κ_2 redefines the method m.

5. For any binary method m with $(m, \forall \alpha \prec \kappa' \ldots) \in \Sigma$ and any $\kappa \prec K \in \Omega$, if $\kappa_1, \kappa_2 \in K$ are independent classes with $[\,] \vdash \kappa_1 \prec \kappa'$ and $[\,] \vdash \kappa_2 \prec \kappa'$, then κ_1 and κ_2 inherit the method m from a common superclass.

The restriction 1 says that the predefined class Obj is the supertype of all ground types. For notational simplicity, we sometimes write $\forall \alpha.\sigma$ for $\forall \alpha \prec Obj.\sigma$.

The restriction 2 implies that if $[\,] \vdash \chi(\overline{\tau_i}) \prec \kappa$ holds then there is exactly one $\forall \overline{\alpha_h \prec \kappa_h} \Rightarrow \chi(\overline{\alpha_h}) \prec \kappa \in \Omega$ such that $[\,] \vdash \tau_i \prec \kappa_i$ for $i = 1, \ldots, h$. The restriction 3 ensures that the method for the same algebraic expressions is not redefined differently. As examples, the programs (a) and (b) in Fig. 5 satisfy the restrictions 2 and 3 but the programs (c) and (d) do not, where we assume that I is a type constructor declared elsewhere.

The restriction 2 does not necessarily mean that if a program contains a subtype declaration for a subclass, then it has to contain a corresponding subtype declaration for each superclass explicitly. If such a subtype declaration is missing for a superclass, we can let a compiler generate one from a corresponding subtype declaration for a subclass. Similarly, if a method redefinition is missing in a subtype declaration according to the restriction 3, we can also let a compiler generate one from a method redefinition in another subtype declaration. In this sense, the program (c) in Fig. 5 can be accepted as a legal program.

(a) **class** A **meth** $m:\sigma = s$	(b) **class** A **meth** $m:\sigma = s$	(c) **class** A **meth** $m:\sigma = s$	(d) **class** A **meth** $m:\sigma = s$
class $B \prec A$	**class** $B \prec A$	**class** $B \prec A$	**class** $B \prec A$ **meth** $m = t$
subtype $I \prec A$ **meth** $m = t'$	**subtype** $I \prec A$	**subtype** $I \prec A$	**subtype** $I \prec A$
subtype $I \prec B$ **meth** $m = t'$	**subtype** $I \prec B$	**subtype** $I \prec B$ **meth** $m = t'$	**subtype** $I \prec B$

Fig. 5 Sample programs for the restrictions 2 and 3

The restriction 2 corresponds to certain restrictions in Haskell (Section 4.3.2 in [14], see also Section 3 in [15]). The restriction 3 coincides with the following conventions in Haskell: (1) If a type is declared to be an instance of a class then it must be declared to be an instance of each of the superclasses; (2) The implementation of a member function declared in a class with respect to a type must be given exactly once, namely in the declaration that declares the type to be an instance of the class. Experiences with Haskell show that the restrictions do not lead to much serious inconvenience in practical programming.

Note that the second condition in the restriction 3 can still be weakened. But the resulting condition would become complicated. Therefore we decided not to do that in the current paper. The restriction 4 is a usual condition in order to ensure the unambiguity of the dynamic binding for methods in the presence of multiple inheritance. As examples, Fig. 6 contains two programs satisfying the restriction.

The restriction 5 is needed to ensure the unambiguity of the dynamic binding for binary methods in the presence of multiple inheritance. To show the problem, consider the program in Fig. 7. If b_1 is a B_1-object and b_2 a B_2-object, then the method eq in the application $eq(b_1 \uparrow Eq, b_2 \uparrow Eq)$ can be bound to different redefinitions in A_1 and in A_2. Thus the dynamic binding is ambiguous. In fact, it is easy to check that the addition of any of the last two declarations to the first three declarations of the program violates the restriction 5.

class A **meth** $x:\sigma = s$	**class** A **meth** $x:\sigma = s$
class $B \prec A$	**class** $B \prec A$ **meth** $x = t$
class $C \prec A$	**class** $C \prec A$
class $D \prec \{B, C\}$	**class** $D \prec \{B, C\}$ **meth** $x = u$

Fig. 6 Sample programs for the restriction 4

class Eq **meth** $eq: \forall \alpha \prec Eq.(\alpha, \alpha) \rightarrow Bool = s$
class $A_1 \prec Eq$ **meth** $eq = s_1$
class $A_2 \prec Eq$ **meth** $eq = s_2$
class $B_1 \prec \{A_1, A_2\}$ **meth** $eq = t_1$
class $B_2 \prec \{A_1, A_2\}$ **meth** $eq = t_2$

Fig. 7 A sample program for the restriction 5

2 Functions and Methods

We define the environment component M to be a set of the forms $x = s$ and $m = \langle \overline{\xi_n \Rightarrow s_n} \rangle$ for recording function bodies and method bodies, respectively. The notation $\langle\overline{\xi_n \Rightarrow s_n}\rangle$, called a *multi-body*, is a finite mapping of ground types ξ_i, called guards, to expressions s_i. A guard ξ_i is intuitively the supertype of all those types, for which the method is declared or redefined with the body s_i. A multi-body is not an expression.

We define an operation $+$ for adding into M a method declaration/redefinition with a given guard as follows:

$$M+(\xi, m=s) \equiv M_1 \cup \{m=\langle\overline{\xi_g \Rightarrow t_g}, \xi \Rightarrow s\rangle\} \quad \text{for } M \equiv M_1 \cup \{m=\langle\overline{\xi_g \Rightarrow t_g}\rangle\}, g \geqslant 0$$

where the case of $g=0$ denotes that M does not contain $m=\ldots$.

We use a judgement of the form $\overline{d_n} \vdash M \rightsquigarrow M'$ with $n \geqslant 1$ to denote that the declarations $\overline{d_n}$ change M into M'.

Figure 8 gives the rules that generate M. Note that the type of the body of each entry in a multi-body can be derived from the guard and the type of the method in the method declaration.

Rule (MethC) deals with a method declaration/redefinition in a class declaration.

$$\frac{(m, \forall\, \alpha \prec \kappa'.\sigma) \in \Sigma \qquad [\,] \vdash \kappa \prec \kappa' \qquad [\,] \vdash s : \forall\, \alpha \prec \kappa.\sigma}{\textbf{class } \kappa \ldots \ \textbf{meth } m[:\forall\, \alpha \prec \kappa.\sigma] = s \vdash M \rightsquigarrow M + (\kappa, m=s)} \quad \text{(MethC)}$$

Rule (MethS) deals with a method redefinition in a subtype declaration.

$$\frac{(m, \forall\, \alpha \prec \kappa'.\sigma) \in \Sigma \qquad [\,] \vdash s : \forall\, \overline{\alpha_h \prec \kappa'_h}.\sigma}{\textbf{subtype } \overline{\alpha_h \prec \kappa'_h} \Rightarrow \chi(\overline{\alpha_h}) \prec \kappa' \ \textbf{meth } m=s \vdash M \rightsquigarrow M + (\chi(\overline{\kappa'_h}), m=s)} \quad \text{(MethS)}$$

Rule (Fun) deals with a function declaration.

$$\frac{[\,] \vdash s : \sigma}{\textbf{fun } x : \sigma = s \vdash M \rightsquigarrow M + (x=s)} \quad \text{(Fun)}$$

Rule (MethAll) deals with a set of declarations.

$$\frac{d_i \vdash M_i \rightsquigarrow M_{i+1} \qquad i = 1, \ldots, n}{\overline{d_n} \vdash M_1 \rightsquigarrow M_{n+1}} \quad \text{(MethAll)}$$

Fig. 8 Generation of the environment component$M$.

We assume that in Fig. 8 the rules contain the environment components Σ and Ω, which have been constructed by a previous pass of the program. This means that a method body can use all classes, type constructors, the subtyping relation, attributes, methods, expression constructors and their types declared in the whole program. A consequence is that recursive methods are implicitly allowed.

In the first two rules we write just one method declaration for notational simplicity. The rules can be easily extended to deal with class and subtype declarations containing zero or more method declarations/redefinitions.

As a simple example, consider

$$(eq, \forall\, \alpha \prec Eq.\,(\alpha, \alpha) \to Bool) \in \Sigma$$

$$[\,] \vdash eqList : \forall\, \alpha_1 \prec Eq.\,(List(\alpha_1), List(\alpha_1)) \to Bool$$

Then the environment component $M \equiv \{eq = \langle Eq \Rightarrow eqEq\rangle\}$ is changed as follows:

$$\textbf{subtype } \alpha \prec Eq \Rightarrow List(\alpha) \prec Eq \ \textbf{meth } eq = eqList$$

$$\vdash M \rightsquigarrow \{eq = \langle Eq \Rightarrow eqEq, List(Eq) \Rightarrow eqList\rangle\}$$

Rule (MethS) needs some explanations. First, we use the type $\chi(\overline{\kappa'_h})$ as the guard for the body s, since $\chi(\overline{\kappa'_h})$ is the least common supertype of all algebraic data types $\chi(\overline{\tau_h})$, where τ_i is a subtype of κ'_i for $i=1, \ldots, h$. Second, the subtype declaration in rule (MethS) is exactly the one for the class κ' with $(m, \forall\, \alpha \prec \kappa'.\sigma) \in \Sigma$; other subtype declarations need not be considered. Lemma 2.1 tells why this suffices.

Let M be the environment component obtained from a program, $m = \langle\overline{\xi_n \Rightarrow s_n}\rangle \in M$ and $(m, \forall\, \alpha \prec \kappa'.\ldots) \in \Sigma$. Then we have the following lemmas.

Lemma 2.1. If $\chi(\overline{\tau_h})$ is a type with $\Gamma \vdash \chi(\overline{\tau_h}) \prec \kappa'$, then there is a least type ξ_f with $1 \leqslant f \leqslant n$ among $\overline{\xi_n}$ satisfying that $\Gamma \vdash \chi(\overline{\tau_h}) \prec \xi_f$. Furthermore, if $\chi(\overline{\tau'_h})$ is another type with $\Gamma \vdash \chi(\overline{\tau'_h}) \prec \kappa'$, then the above ξ_f is still the least type among $\overline{\xi_n}$ satisfying that $\Gamma \vdash \chi(\overline{\tau'_h}) < \xi_f$.

Proof. By the assumption that $\Gamma \vdash \chi(\overline{\tau_h}) < \kappa'$, and the restriction 2 in Section 1.9, there must be a subtype declaration for $\overline{\alpha_h < \kappa'_h} \Rightarrow \chi(\overline{\alpha_h}) < \kappa' \in \Omega$. If the subtype declaration contains a redefinition of m, then by rule (MethS), we know that $\xi_f = \chi(\overline{\kappa'_h})$ is the unique type among $\overline{\xi_n}$, which is of the form $\chi(\ldots)$. Thus the assertion of the lemma holds. If the subtype declaration does not contain a redefinition of m, then by the restriction 3 in Section 1.9, the method m of the class κ' is the inherited method in all subtype declarations for χ and κ with $[\,] \vdash \kappa < \kappa'$. By the restriction 4 in Section 1.9 the method m in κ' is redefined in κ' or inherited from one unique class. Thus the assertion of the lemma holds. □

Lemma 2.2. For any type τ with $\Gamma \vdash \tau < \kappa'$, there is a least type ξ_f among $\overline{\xi_n}$ satisfying that $\Gamma \vdash \tau < \xi_f$.

Proof. Assume that the type τ is a class. Then the assertion of the lemma follows from the restriction 4 in Section 1.9. If τ is an algebraic data type $\chi(\overline{\tau_h})$, then by Lemma 2.1, the assertion of the lemma holds. □

3 Reduction

The computation of the expressions is given by a reduction relation of the form $\Gamma \vdash s \rhd t$ defined by a set of single-step reductions and a set of context reductions. The context reductions are standard. We consider only the single-step reductions here.

In order to formulate the reductions compactly, we assume that a form $s \uparrow \tau$ on the left-hand side of a reduction always stands for $s \uparrow \tau_1 \ldots \uparrow \tau_n \uparrow \tau$, $n \geqslant 0$, or s itself in an environment Γ with $\Gamma \vdash s : \tau$.

Figure 9 gives all single-step reductions. The $\lambda\beta$-reduction generalizes the standard one $\Gamma \vdash (\lambda x : \tau . s\ t) \rhd s[t/x]$. Note that due to nonvariance, the domain type of a function type always remains unchanged along the subtyping relation. The $\Lambda\beta$- and let-reduction are standard. The fun-replacement is trivial. The get-simplification considers the access of an attribute in an object with casting operators. But this reduction is not that interesting, since we do not consider the manipulation of objects in this paper. The $\uparrow$-reduction is also trivial, but without it the formulation of the argument $(\overline{v_n \uparrow \tau})$ in the $\langle\rangle$-reduction would become a little more complex.

$\lambda\beta$-reduction $\Gamma \vdash (((\lambda x : \tau . s) \uparrow \tau \to \tau')t) \rhd s[t/x] \uparrow \tau'$

$\Lambda\beta$-reduction $\Gamma \vdash (\Lambda\alpha < \kappa . s[\tau]) \rhd s[\tau/\alpha]$

fun-replacement $\Gamma \vdash x \rhd s$, where $x = s \in M$

let-reduction $\Gamma \vdash$ let $x : \sigma = s$ **in** $t \rhd t[s/x]$

get-simplification $\Gamma \vdash (o_\kappa \uparrow \kappa').a \rhd o_\kappa.a$

$\uparrow$-reduction $\Gamma \vdash (\overline{s_n}) \uparrow (\overline{\tau_n}) \rhd (\overline{s_n \uparrow \tau_n})$

$\langle\rangle$ reduction $\Gamma \vdash ((m[\tau, \ldots] \uparrow (\overline{\tau}) \to \tau')\ (\overline{v_n \uparrow \tau})) \rhd (s_f[\overline{\tau'_h}, \ldots]\ (\overline{v_n \uparrow \tau_\mu})) \uparrow \tau'$,

where $m = \langle \overline{\xi_g \Rightarrow s_g} \rangle \in M$, $\Gamma \vdash v_i : \tau_i$ for $i = 1, \ldots, n$, $\tau_\mu \equiv \mu(\{\overline{\tau_n}\}, \tau)$, ξ_f is the least type among $\overline{\xi_g}$ satisfying that $\Gamma \vdash \tau_\mu < \xi_f$, and if $\xi_f \equiv \chi(\ldots)$ then $\overline{\tau'_h}$ satisfies that $\chi(\overline{\tau'_h}) \equiv \tau_\mu$, otherwise $\overline{\tau'_h} \equiv \tau_\mu$.

Fig. 9 The single-step reductions

The $\langle\rangle$-reduction formalizes the dynamic binding for methods. It is the only rule that introduces a multi-body into the computation. Roughly, the $\langle\rangle$-reduction works as follows. First, the first argument is reduced to a tuple of values. Then a certain common type of these values is computed. Finally the type determines a body. The existence of such a body is ensured by the static type inference. The context conditions (Section 1.9) ensure the uniqueness. If you remove all casting operators then the reduction becomes

$$\Gamma \vdash (m[\tau, \ldots]\ (\overline{v_n})) \rhd (s_f[\overline{\tau'_h}, \ldots](\overline{v_n}))$$

The definition of the function $\mu(\{\overline{\tau_n}\}, \tau)$ is the key to compute a common type of the values in the first argument tuple:

$$\mu(\{\ldots\}, \chi(\overline{\tau_h})) = \chi(\overline{\tau_h})$$

$$\mu(\{\overline{\kappa_n}\},\kappa) = \text{a minimal class } \kappa_\mu \text{ with } [\,] \vdash \kappa_i \prec \kappa_\mu,\ i = 1,\ldots,n,\ [\,] \vdash \kappa_\mu \prec \kappa$$

$$\mu(\{\chi(\overline{\tau_h^1}),\ldots,\chi(\overline{\tau_h^n})\},\kappa) = \chi(\overline{\kappa_h}) \quad \text{for } \overline{\alpha_h \prec \kappa_h} \Rightarrow \chi(\overline{\alpha_h}) \prec \kappa \in \Omega$$

$$\mu(\ldots\text{otherwise}\ldots,\kappa) = \kappa$$

An easy observation is that except the second case the time complexity of μ is linear to the size of the input. The first case in μ enables some type-based optimization of method applications.

The additional type arguments $\overline{\tau'_h}$ are needed in the case that $\xi_f \equiv \chi(\ldots)$, since the body s_f in that case stems from a subtype declaration $\overline{\alpha_h \prec \kappa'_h} \Rightarrow \chi(\overline{\alpha_h}) \prec \kappa'$ and is a type abstraction of the form $\Lambda\,\overline{\alpha_h \prec \kappa'_h}.\ldots$.

Assume the program in Figs. 2, 3 and 4. Let M contain

$$eq = \langle Eq \Rightarrow eqEq,\ Point \Rightarrow eqPoint,\ ColorPoint \Rightarrow eqColorPoint,\ List(Eq) \Rightarrow eqList \rangle$$

Then the following reduction steps hold:

$$eq[List(Eq)](con(o_{\text{Point}},empty) \uparrow List(Eq),\ con(o'_{\text{Point}},empty) \uparrow List(Eq))$$
$$\rhd (eqList[Eq](\ldots\text{the same argument as above}\ldots)) \uparrow Bool$$
$$eq[Eq](o_{\text{Point}},\ o'_{\text{ColorPoint}} \uparrow Point)$$
$$\rhd (eqPoint[Point](\ldots\text{the same argument as above}\ldots)) \uparrow Bool$$
$$eq[Eq](con(o_{\text{Point}},empty) \uparrow Eq,\ con(o'_{\text{Point}},empty) \uparrow Eq)$$
$$\rhd (eqList[Eq](\ldots\text{the same argument as above}\ldots)) \uparrow Bool$$
$$eq[Eq](o_{\text{Point}} \uparrow Eq,\ con(o'_{\text{Point}},empty) \uparrow Eq)$$
$$\rhd (eqEq[Eq](\ldots\text{the same argument as above}\ldots)) \uparrow Bool$$

The following lemma states the main properties of the function μ:

Lemma 3.1. Assume $m = \langle \overline{\xi_g \Rightarrow s_g} \rangle \in M$ and $\Gamma \vdash \tau_i \prec \tau$ hold for $i=1,\ldots,n$.

1. Then $\mu(\{\overline{\tau_n}\},\tau)$ always yields a type τ_μ satisfying that $\Gamma \vdash \tau_\mu \prec \tau$, and
2. if $(m,\forall\,\alpha \prec \kappa'.\ldots) \in \Sigma$ and $\Gamma \vdash \tau \prec \kappa'$, then there is a least type ξ_f with $1 \leqslant f \leqslant g$ among $\overline{\xi_g}$ satisfying that $\Gamma \vdash \tau_\mu \prec \xi_f$.

Proof. 1. It is easy to check that for each $\mu(\{\overline{\tau_n}\},\tau)$, exact one pattern in the definition of μ is applicable. The property that $\Gamma \vdash \tau_\mu \prec \tau$ follows directly from the definition of μ.

2. It follows directly from Lemma 2.2. □

Although the type τ_μ yielded by $\mu(\{\overline{\tau_n}\},\tau)$ is not necessarily the least or a minimal type satisfying that $\Gamma \vdash \tau_i \prec \tau_\mu$ for $i=1,\ldots,n$ and $\Gamma \vdash \tau_\mu \prec \tau$, it is small enough to determine a body in the dynamic binding. Formally we state this as the following theorem.

Theorem 3.2. Assume the notations in Lemma 3.1. Then for any type τ'_μ such that $\Gamma \vdash \tau'_\mu \prec \tau_\mu$ and $\Gamma \vdash \tau_i \prec \tau'_\mu$ for $i=1,\ldots,n$, the type ξ_f is the least type among $\overline{\xi_g}$ satisfying that $\Gamma \vdash \tau'_\mu \prec \xi_f$.

Proof. We check each pattern in the definition of μ.

Let the pattern be $\mu(\{\ldots\},\chi(\overline{\tau_h}))$. Then the assertion follows from Lemma 2.1.

Let the pattern be $\mu(\{\overline{\kappa_n}\},\kappa)$. If m is a simple method, then $\tau_\mu \equiv \tau'_\mu$, thus the assertion holds trivially. If m is a binary method, then the assertion follows from the restriction 5.

Let the pattern be $\mu(\{\chi(\overline{\tau_h^1}),\ldots,\chi(\overline{\tau_h^n})\},\kappa)$. Then the assertion follows from Lemma 2.1.

The pattern $\mu(\ldots\text{otherwise}\ldots,\kappa)$ can be divided into two cases: $\mu(\{\kappa_j,\chi(\overline{\tau_h}),\ldots\},\kappa)$ and $\mu(\{\chi(\overline{\tau_h}),\chi'(\overline{\tau_g}),\ldots\},\kappa)$. In the former case, the assertion follows from the restriction 2 and the first condition of the restriction 3. In the latter case, the assertion follows from the restriction 2 and the second condition of the restriction 3. □

For simple methods one may expect that the specialization of the $\langle\rangle$-reduction is equivalent to the following intuitive reduction:

$$\Gamma \vdash ((m[\tau,\ldots] \uparrow \tau \to \tau')(v \uparrow \tau)) \rhd (s_f[\overline{\tau'_h},\ldots]v) \uparrow \tau',$$

where τ_μ satisfies that $\Gamma \vdash v : \tau_\mu$, ξ_f and $\overline{\tau'_h}$ are computed as in the $\langle\rangle$-reduction. Strictly speaking, both reductions are different in that the type τ_μ yielded by the function μ in the $\langle\rangle$-reduction may be bigger than the type τ_μ in the above intuitive reduction. However, Theorem 3.2 ensures that the results of both reductions are equivalent to the casting operators.

It is straightforward to prove the type-preserving property of the reductions.

Theorem 3.3. If $\Gamma \vdash s : \tau$ and $\Gamma \vdash s \rhd t$ then $\Gamma \vdash t : \tau$.

Proof. The type-preserving properties of the $\lambda\beta$-, $\Lambda\beta$, let and $\uparrow$ reductions and the fun-replacement are straightforward. The type-preserving property of the $\langle\rangle$-reduction follows from Lemma 3.1-1 and the restriction that in the type $\forall\ \alpha \prec \kappa.\forall\ \overline{\alpha_n \prec \kappa_n}.(\bar{\alpha}) \to \tau'$ of a method, the type variable α does not occur within the domain type of a function type in τ'. □

4 Related Work

The goal of FOC is to model the combination of a number of important concepts in the mainstream object-oriented and functional languages. Abadi and Cardelli investigated several relevant object calculi[16]. Our work could be seen as another effort in the same line.

Pizza[1] extends Java by parameterized classes, algebraic data types and higher-order functions. The extensions are based on the F-bounded polymorphism[17], which is quite different from the concepts in the mainstream languages. Two concrete things that FOC has but Pizza does not are the support of a unified concept for basic and constructed types like in a functional language and a mechanism for binary methods, for which you do not have to pay if you do not use. Parameterized classes are missing in FOC, but we do not see any serious difficulties in integrating it into FOC in certain way.

Objective ML[2] is based on an extension of ML-polymorphism by polymorphic access to record types and preserves most good properties of the ML-polymorphism. Similarities between FOC and Objective ML are, for example, explicit casting conversions and type preservation in method redefinitions. One difference between FOC and Objective ML is that FOC unifies Haskell classes and object-oriented classes, whereas Objective ML does not.

Object SML[3,4] introduces extended ML datatypes for classes and treats objects as values of these extended ML datatypes. Similar to Objective ML, Object SML does not provide a unified concept for Haskell and object-oriented classes. In addition, Object SML does not allow binary methods.

Haskell++[5] extends Haskell with objects based on the theories of existential types (see Refs. [18~21]. Although existential types can formulate the relationship between interfaces and classes, they have problems in formulating the general concept of inheritance.

In order to assure static type safety for binary methods, Bruce *et al.*[22] introduced the notion of match in the languages PolyTOIL[4] and LOOM[23]. Match is a generalization of subtyping between object classes. PolyTOIL and LOOM have the same property as FOC that binary methods are statically type-safe. PolyTOIL combines subtyping, match, LOOM match and an unusual notion called hash types to achieve some benefits of subtyping. Thus the mechanisms for binary methods in PolyTOIL and LOOM are unusual. Bruce's binary methods distinguish between the receiver and other arguments as usual, whereas in our case a number of arguments play the same part in the dynamic binding.

Castagna, Ghelli and Longo[24] were the first, who combined subtyping, multibodies and dynamic binding in the method reduction. The $\langle\rangle$-reduction in FOC has some similarities to their method reduction. In their calculus the entries of a multibody need not have a common type scheme. This may lead to a type system that can yield a very large set of types as a type annotation.

The ⟨⟩-reduction in FOC has similarities to a dynamic binding reduction in the calculus TOFL[25]. But the TOFL approach is quite different and much more complicated than FOC. First, TOFL does not include explicit casting operators. In order to define the dynamic binding reduction, TOFL imposes some additional restrictions to ensure the existence and uniqueness of a least type of a term and includes an additional type inference system to effectively derive the least type. FOC includes explicit casting operators and thus does not need these properties and the additional type inference system any more. Indeed, these properties do not help in FOC, and the restrictions in Section 1.9 are needed anyway. Second, TOFL defines the concept of values as irreducible terms, while FOC defines the concept completely syntactically, contributes much to the simplicity and clarity of the work and provides the possibility to use different reduction strategies in the implementation. Third, the restrictions on the syntax of TOFL are quite different and to a great extent not so clear as those on the syntax of FOC. Fourth, the reductions in TOFL are much less efficiently implementable than the reductions of FOC. As an example, the computation of μ in FOC does not need the inner structure of the types, whereas that of the corresponding function in the dynamic binding reduction in TOFL does.

Parallel to the work reported here we designed a small language based on FOC[26]. The current prototyping implementation also implements a type reconstruction for a special case of the current version of FOC based on an algorithm in Ref. [27].

5 Conclusion and Future Directions

We have presented the core language FOC, which combines quite a number of important concepts in Eiffel, Java, ML and Haskell. Although the explicit casting operators may make programs look a little bit involved, the approach is to a great extent simple, natural and straightforward. The approach is also close to the practical implementations that use type tags.

It is interesting to mention a few specializations of FOC to show why "you pay only for what you use". If we have only functional programs, then we need no casting operators. In this case subtype declarations model some aspects of Haskell instance declarations. If we have only object-oriented programs, which contain only classes and basic types, then we need no subtype declarations and can assume that casting operators are implicitly everywhere in an expression. In fact, the casting operators now correspond to the usual checking of subtyping between classes. If we do not have multiple inheritance, then the restrictions 4 and 5 in Section 1.9 hold automatically. If we do not have binary methods, then the restriction 5 holds automatically.

We are currently working on an implementation of FOC, which extends Java. We want to point out that there is a well-known problem with static type safety in the presence of assignment and co-variant subtyping for constructed types. Following what Java does with arrays, we allow assignments to components of algebraic types and live with static type unsafety.

References

1 Odersky M, Wadler P. Pizza into Java: Translating theory into practice. In: Proceedings of the 24th ACM Symp., Principles of Programming Languages. 1997. 146~159

2 Rémy D, Vouillon J. Objective ML: a simple object-oriented extension of ML. In: Proceedings of the 24th ACM Symp. Principles of Programming Languages. 1997. 40~53

3 Reppy J, Riecke J. Classes in object ML via modules. In: Proceedings of 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation. SIGPLAN Notices, 1996, 31(5): 171~180

4 Reppy J, Riecke J. Classes in object ML via modules. Technical Report, 1996, Presented at the FOOL'3 workshop
5 Hughes J, Sparud J. Haskell++: An object-oriented extension of Haskell. In: Proceedings of 1995 Workshop on Haskell, 1995
6 Hoang M, Mitchell J. Lower bounds on type inference with subtypes. In: Proceedings of the 22nd ACM Symp. Principles of Programming Languages. 1995. 176～185
7 Aiken A, Wimmers E. Type inclusion constraints and type inference. In: Proceedings of the Functional Programming Languages and Computer Architecture. ACM, 1993. 31～41
8 Eifrig J, Smith S, Trifonov V *et al*. An interpretation of typed OOP in a language with state. Journal of Lisp and Symbolic Computation, 1995. 357～397
9 Smith G S. Principal type schemes for functional programs with overloading and subtyping. Science of Computer Programming, 1994,23:197～226
10 Wand M. Complete type inference for simple objects. In: Proceedings of the 2nd IEEE Symp. Logic in Computer Science, 1987, 37～44, Corrigendum in Proceedings of the 3rd IEEE Symp. Logic in Computer Science
11 Rémy D. Typechecking records and variants in a natural extension of ML. In: Proceedings of the 16th ACM Symp. Principles of Programming Languages, 1989. 77～87
12 Ohori A. A compilation method for ML-style polymorphic record calculi. In: Proceedings of the 19th ACM Symp. Principles of Programming Languages, 1992. 154～165
13 Harper R, Mitchell J. The essence of ML. In: Proceedings of the 15th ACM Symp. Principles of Programming Languages, 1988. 28～46
14 Hudak P, Peyton Jones S, Wadler P. Report on the Programming Language Haskell: a Non-strict. Purely functional language (Version 1.2), ACM SIGPLAN Notices, 1992,27(5)
15 Nipkow T, Prehofer C. Type reconstruction for type classes. Journal of Functional Programming, 1995,5(2):201～224
16 Abadi M, Cardelli L. A Theory of Objects. Springer-Verlag, 1996
17 Canning P, Cook W, Hill W *et al*. F-bounded polymorphism for object-oriented programming. In: Proceedings of the Functional Programming Languages and Computer Architecture, 1989. 273～280
18 Mitchell J, Plotkin G. Abstract types have existential type. ACM Transactions on Programming Language and System, 1988,10(3):475～502
19 Cardelli L, Wegner P. On understanding types, data abstraction, and polymorphism. Computing Surveys, 1985,17(4): 471～522
20 Äufer K. Combining type classes and existential types. In: Proceedings of the Latin American Informatics Conference (PANEL), ITESM-CEM, Mexico, 1994
21 Äufer K, Odersky M. Polymorphic type inference and abstract data types. ACM Transactions on Programming Languages and Systems, 1994
22 Bruce K, Schuett A, Gent R V. PolyTOIL: A type-safe polymorphic object-oriented language (extended abstract). In: Proceedings of the 9th European Conference on Object-Oriented Programming, Springer-Verlag LNCS 952, 1995. 27～51
23 Bruce K, Peterson L, Fiech A. Subtyping is not a good "match" for object-oriented languages. In: Proceedings of the 11th European Conference on Object-Oriented Programming, Springer-Verlag LNCS 1241, 1997
24 Castagna G, Ghelli G, Longo G. A calculus for overloaded functions with subtyping. Information and Computation, 1995, 117(1):115～135
25 Qian Z, Krieg-Brückner B. Object-Oriented functional programming with late binding. In: Cointe P ed, Proceedings of the 10th European Conference on Object-Oriented Programming, Springer-Verlag LNCS 1098, 1996, 48～72. Long version as technical report FB Informatik, Universität Bremen, 1996
26 Qian Z, Abd Moulah B. Entwurf und prototypische implementierung einerobjek torientierten funktionalen sprache. In: Tagungsband der GI-Jahrestagung 1997, Springer Verlag "Informatik Aktuell", 1997
27 Qian Z, Krieg-Brückner B. Object-Oriented functional programming and type reconstruction. In: Haveraaen M, Owe O, Dahl O-J eds. Recent Trends in Data Type Specification, Springer-Verlag LNCS 1130, 1996. 458～477

28 Curien P-L, Ghelli G. Coherence of subsumption, minimum typing and the type checking in $F_{\leqslant}$. Mathematical Structures in Computer Science, 1992,2(1)

Appendix A Formal Treatment of the Types and Terms

A formal calculus for FOC can be obtained from the calculus $F_{\leqslant}$[28] by stratifying the types into monomorphic and polymorphic types, replacing implicit coercion by explicit type lifting and adding type constructors, classes, attributes, multi-bodies and a dynamic binding mechanism.

A.1 The environment

It is assumed that the set of type variables and the set of expression variables are always disjoint. We define $\mathcal{FV}(O)$ to be the set of all free (type or expression) variables in a syntactic object O, i.e.,

$$\mathcal{FV}(\tau) = \text{the set of all type variables in } \tau$$
$$\mathcal{FV}(\forall\, \alpha \prec \kappa.\sigma) = \mathcal{FV}(\sigma) - \{\alpha\}$$
$$\mathcal{FV}(\Gamma) = \bigcup_{\alpha\prec\kappa\in\Gamma}\{\alpha\} \cup \bigcup_{x:\sigma\in\Gamma}(\{x\} \cup \mathcal{FV}(\sigma))$$

An environment Γ must satisfy the least relation $\vdash \Gamma$ closed under the rules in Fig. 10. Note that the judgment of the form $\Gamma \vdash \sigma$ used in the rules is defined in the next subsection.

$$\frac{}{\vdash [\,]}([\,]\text{-env}) \qquad \frac{\vdash \Gamma \quad \alpha \notin \mathcal{FV}(\Gamma) \quad \kappa \text{ occurs in } \Omega}{\vdash \Gamma + [\alpha \prec \kappa]}(\prec\text{-env}) \qquad \frac{\vdash \Gamma \quad x \notin \mathcal{FV}(\Gamma) \quad \Gamma \vdash \sigma}{\vdash \Gamma + [x:\sigma]}(:\text{-env})$$

Fig. 10 Well-formedness of environments

Intuitively, rule ([]-env) creates an empty environment, rules ($\prec$-env) and (:-env) ensure that each (expression or type) variable in an environment may only be declared at most once, and that no variable may be used before it is declared. Note that it suffices to have only classes as bounds in the environment Γ, since it is so in the source programs.

A.2 Well-formed types

The inference rules for environments are based on the well-formedness of types. Formally, the relation $\Gamma \vdash \sigma$, i.e. σ is *well-formed* under Γ, is the least relation closed under the rules in Fig. 11. It is easy to prove that if σ is well-defined under Γ then all its free variables are declared in Γ.

$$\frac{\alpha \prec \kappa \text{ is an element of } \Gamma}{\Gamma \vdash \alpha}(\alpha\text{-type}) \qquad \frac{\kappa \text{ occurs in } \Omega}{\Gamma \vdash \kappa}(\kappa\text{-type})$$

$$\frac{\Gamma \vdash \tau_i (i=1,\ldots,h) \quad \chi \text{ is } h\text{-ary and in } \Omega}{\Gamma \vdash \chi(\overline{\tau_h})}(\chi\text{-type})$$

$$\frac{\Gamma + [\alpha \prec \kappa] \vdash \sigma}{\Gamma \vdash \forall\, \alpha \prec \kappa.\sigma}(\sigma\text{-type})$$

Fig. 11 Well-formedness of types

A.3 The subtyping relation

A set of inference rules for deriving $\Gamma \vdash \sigma \prec \sigma'$ is given in Fig. 12.

$$\frac{\alpha \prec \kappa \text{ is an element of } \Gamma}{\Gamma \vdash \alpha \prec \kappa}(\alpha\text{-subty}) \qquad \frac{\kappa \prec K \in \Omega \quad \kappa' \in K}{\Gamma \vdash \kappa \prec \kappa'}(\kappa\text{-subty})$$

$$\frac{\forall\, \overline{\alpha_h \prec \kappa_h} \Rightarrow \chi(\overline{\alpha_h}) \prec \kappa \in \Omega \quad \Gamma \vdash \tau_i \prec \kappa_i (i=1,\ldots,h)}{\Gamma \vdash \chi(\overline{\tau_h}) \prec \kappa}(\chi\text{-}\kappa\text{-subty})$$

$$\frac{\Gamma \vdash \tau_i \prec \tau'_i (i=1,\ldots,h) \quad \chi \text{ is not} \rightarrow}{\Gamma \vdash \chi(\overline{\tau_h}) \prec \chi(\overline{\tau'_h})}\chi\text{-subty})$$

$$\frac{\Gamma \vdash \tau_1 \prec \tau_2 \quad \Gamma \vdash \tau}{\Gamma \vdash \tau \rightarrow \tau_1 \prec \tau \rightarrow \tau_2}(\rightarrow\text{-subty})$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \prec \tau}(\text{refl-subty}) \qquad \frac{\Gamma \vdash \tau_1 \prec \tau_2 \quad \Gamma \vdash \tau_2 \prec \tau_3}{\Gamma \vdash \tau_1 \prec \tau_3}(\text{trans-subty})$$

$$\frac{\Gamma + [\overline{\alpha_n \prec \kappa_n}] \vdash \tau \prec \tau'}{\Gamma \vdash \forall\, \overline{\alpha_n \prec \kappa_n}.\tau \prec \forall\, \overline{\alpha_n \prec \kappa_n}.\tau'}(\text{poly-subty})$$

Fig. 12 The subtyping relation

The first three rules need no further explanations. Rule (χ-subty) says that the subtyping relation is co-variant for all type constructors except $\rightarrow$. Rule ($\rightarrow$-subty) says that the subtyping relation is co-variant for $\rightarrow$ at the range position and non-variant at the domain position.

There is at least one reason for us to adopt non-variance, instead of the usual contra-variance, at the domain position of a function type. If a programmer writes a subtype declaration $\alpha \prec A \Rightarrow (\alpha \rightarrow B) \prec C$, then he would expect that $\Gamma \vdash (\tau \rightarrow B) \prec C$ holds for all types τ satisfying that $\Gamma \vdash \tau \prec A$. The contra-variance would imply that $\Gamma \vdash (\tau' \rightarrow B) \prec C$ for all types τ' satisfying that $\Gamma \vdash \tau \prec \tau'$ and $\Gamma \vdash \tau \prec A$, i.e. $\Gamma \vdash (\tau'' \to B) \prec C$ for all types τ'' satisfying that $\Gamma \vdash A \prec \tau''$. This is usually not what the programmer wants with the original subtype declaration.

Proposition A.1. Let Γ be an environment and σ and σ' arbitrary types such that $\Gamma \vdash \sigma$ and $\Gamma \vdash \sigma'$ hold. Then it is decidable whether $\Gamma \vdash \sigma \prec \sigma'$ holds.

A.4 Typing expressions

We use Δ to denote the set of the pairs $(\kappa, \{\overline{a_n : \xi_n}\})$ for all classes κ and all attribute declarations $\overline{a_n : \xi_n}$ in κ. If $(\kappa, \{\overline{a_n : \xi_n}\}) \in \Delta$ then we use $\Delta(\kappa)$ to denote $\{\overline{a_n : \xi_n}\}$.

We use Ξ to denote the set of all value constructors $c : \forall\, \overline{\alpha_n}.\, (\overline{\tau_h}) \to \chi(\overline{\alpha_n})$.

We use Φ to denote the set of the forms $x : \sigma$ for all function declarations **fun** $x : \sigma = s$.

Now an environment contains Ω, Σ, Δ, Ξ, Φ and Γ. Since Ω, Σ, Δ, Ξ and Φ are fixed in a given program, we will omit them in the inference rules.

Let $\Gamma \vdash t : \sigma$ denote that an expression t has a type σ under an environment Γ. Then the relation can be defined as the least relation closed under the rules in Fig. 13.

$$\frac{x : \sigma \in \Gamma \cup \Phi}{\Gamma \vdash x : \sigma}(\text{VTaut}) \quad \frac{(\text{m} : \sigma, \kappa) \in \Sigma}{\Gamma \vdash \text{m} : \sigma}(\text{MTaut}) \quad \frac{c : \forall\, \overline{\alpha_n}.\, \tau \in \Xi \quad \Gamma \vdash \tau_i (i = 1, \ldots, n)}{\Gamma \vdash c : \tau[\overline{\tau_n / \alpha_n}]}(\text{CTaut})$$

$$\frac{\Gamma \vdash s : \forall\, \alpha \prec \kappa.\, \sigma \quad \Gamma \vdash \tau \prec \kappa}{\Gamma \vdash s[\tau] : \sigma[\tau/\alpha]}(\text{Tapp}) \quad \frac{\Gamma + [\alpha \prec \kappa] \vdash s : \sigma}{\Gamma \vdash \Lambda\alpha \prec \kappa.\, s : \forall\, \alpha \prec \kappa.\, \sigma}(\text{Tabs})$$

$$\frac{\Gamma + [x : \tau] \vdash s : \tau'}{\Gamma \vdash \lambda x : \tau.\, s : \tau \to \tau'}(\text{Abs}) \quad \frac{\Gamma \vdash s : \tau' \to \tau \quad \Gamma \vdash t : \tau'}{\Gamma \vdash (s\ t) : \tau}(\text{App})$$

$$\frac{\Gamma \vdash s : \sigma \quad \Gamma + [x : \sigma] \vdash t : \tau}{\Gamma \vdash \textbf{let } x : \sigma = s \textbf{ in } t : \tau}(\text{Let})$$

$$\frac{\Gamma \vdash s : \tau \quad \Gamma \vdash \tau \prec \tau'}{\Gamma \vdash (s \uparrow \tau') : \tau'}(\text{Sub})$$

$$\frac{\Gamma \vdash \kappa}{\Gamma \vdash o_\kappa : \kappa}(\text{Obj}) \quad \frac{\Gamma \vdash s : \kappa \quad a : \xi \in \Delta(\kappa)}{\Gamma \vdash s.a : \xi}(\text{Attr})$$

Fig. 13　Typing rules in FOC

Rule (Sub) deals with explicit type liftings. The last two rules are straightforward. All other rules are standard in typed calculi. Note that rules (Sub), (Abs), (App) and (Attr) require that the expressions should be of monomorphic types. Thisensures that a polymorphic function/method must be first instantiated into a monomorphic expression by rule (Tapp) and then applied to another monomorphic expression.

结合面向对象和函数式语言的概念

QIAN Zhen-yu　Besma Abd Moulah

(Universität Bremen　Germany)

摘要　考虑了结合面向对象和函数式程序风范的问题.与这一方向的大多数方法相比,这种结合方法有下面两个优点:首先,结合了在广泛流行的几种主语言中非常有名的一些重要概念.换言之,没有引入新的语言概念并试图以新的概念为基础解释众所周知的语言概念.其次,这种结合具有下面的性质:如果整个语言以传统方式使用则不受个别的语言概念的影响,这样,只有在使用一个语言概念的时候才需要关注它.具体地说,提出了一个具有简明操作语言的用于函数式面向对象程序设计的核心语言,它具有如上所述的性质.这个核心语言结合了Eiffel,Java,ML和Haskell语言中的下列核心语言概念:对象,类,多重继承,方法重定义,动态绑定,静态类型安全性,二元方法,代数数据类型,高阶函数,ML-多态性.

关键词　面向对象程序设计,多重集成,方法重定义,动态绑定,静态类型安全性,二元方法,函数式程序设计,代数数据类型,高阶函数,ML-多态性.

中图法分类号　TP311