STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLVI**, Number 1, 2001

PURELY FUNCTIONAL PROGRAMMING AND THE OBJECT-ORIENTED INHERITANCE AND POLYMORPHISM

LEHEL KOVÁCS, GÁBOR LÉGRÁDI, AND ZOLTÁN CSÖRNYEI

Abstract. According to the purely functional paradigm, the value of an expression depends only on the values of its subexpressions, if any.

In this paper we introduce this principle in the object-oriented paradigm. The simplicity and power of functional languages is due to properties like pure values, first-class functions, and implicit storage management. We must extend these properties with a strong type-system.

The values must be typed, the type system used for this purpose is the higher-order, explicitly-typed, polymorphic λ-calculus with subtyping, called $F^{\omega}_{\leq}$.

This type-system must be prepared for basic mechanisms of object-oriented programming: encapsulation, message passing, subtyping and inheritance. Polymorphic functions arise naturally when lists are manipulated and lists with elements of any types can be accomplished by a straightforward generalization of inheritance.

Interesting questions are also, how to introduce the object- oriented inheritance, the subtyping mechanism and the object oriented polymorphism.
Key Words and Phrases: untyped and typed λ-calculus, object-oriented programming, inheritance, polymorphism.

1. Introduction

In this paper we introduce the principle of purely functional paradigm into the object-oriented paradigm and we concentrate the solutions of problems related to each kind of inheritance and polymorphism.

We are going to focus our attention on a simple object model in which an object has a value that can be modified by messages.

An object has an internal state and methods, the states have unusual feature: the internal state of all objects is immutable, the result of methods are values rather side-effects of variables.

The simplicity and power of functional languages are due to properties like pure values, first-class functions, and implicit storage management. We must extend

2000 *Mathematics Subject Classification.* 68P05.

1998 *CR Categories and Descriptors.* D.1.1 [**Software**]: Programming Techniques – *Applicative (Functional) Programming*; D.1.5 [**Software**]: Programming Techniques – *Object-oriented Programming.*

these properties with a strong type system, the type system $F^{\omega}_{\leq}$ is applied. A summary of this type system is given in the next section.

There are three object models from literature, the record model, the existential-type model and the axiomatic model [2]. We use the the record model to refer to the representation of objects by recursively defined records in which a class is represented by a record of variables and methods. Using λ-calculus and type system $F^{\omega}_{\leq}$ extended by record structures we have a purely functional model of object-oriented programming.

We also show how to write well-typed polymorphic functions that operates on different objects. A polymorphic function can be applied to arguments of more than one type. We concentrate on parametric polymorphism, a special kind of polymorphism, in which type expressions are parametrized.

2. Type system $F^{\omega}_{\leq}$

Church's ordinary typed λ-calculus has the name F_1, F_2 correspond to Girard's and Reynold's second-order typed λ-calculus. F_3 is obtained from F_2 by allowing type constructors that transform existing types into new types, and it follows that the *kind* has the form $*$ or $* \rightarrow kind$. Using successively higher kind, we obtain systems F_4, F_5,... The union of all these systems is called F^{ω}. In system F^{ω} the syntax of *kinds* has the form

$$\begin{array}{lcl} kind & := & * \\ & | & kind \rightarrow kind \end{array}$$

One of the natural extension of type system F^{ω} deals with subtyping, a simple version of such a system is $F^{\omega}_{\leq}$. The description of this system is described for example in [3].

A further extension of $F^{\omega}_{\leq}$ introduces *records* [5], it is mainly used to formulate models for object-oriented systems [1,4].

The syntax of record type, record term construction and record term selection are as follows:

$$\begin{array}{lcl} < type > & := & \{| < name_1 >:< type_1 >, \ldots, < name_n >:< type_n > |\} \\ < term > & := & \{< name_1 >=< term_1 >, \ldots, < name_n >=< term_n >\} \\ & | & < term > . < name > \end{array}$$

Now we will extend the rule-system $F^{\omega}_{\leq}$ to allow records.

(1) The *kinding* rule for record:

$$\Gamma \vdash < type_i > \in * \text{ for each } i$$
$$\Rightarrow \Gamma \vdash \{| < name_1 >:< type_1 >, \ldots, < name_n >:< type_n > |\} \in *$$

(2) The rule of record *introduction*:

$$\Gamma \vdash < term_i >:< type_i > \quad \text{for each } i$$
$$\Rightarrow \Gamma \vdash \{< name_1 >=< term_1 >, \ldots, < name_n >=< term_n >\}$$
$$: \{| < name_1 >:< type_1 >, \ldots, < name_n >:< type_n > |\}$$

(3) The rule of record *elimination* for the record $< term >\equiv \{< name_1 >=< term_1 >, \ldots, < name_n >=< term_n >\}$:

$$\Gamma \vdash < term >: \{| < name_1 >:< type_1 >, \ldots, < name_n >:< type_n > |\}$$
$$\Rightarrow < term > . < name_i >:< type_i >$$

(4) And finally, the *subtype* rule has two assumptions,
- the subtype record has at least the same fields as the other record type,
- each of the types of the fields of the subtype need to be subtypes of the types of the corresponding fields (if they exist) in the other type.

$$\{name_{1,1}, \ldots, name_{1,m}\} \supseteq \{name_{2,1}, \ldots, name_{2,n}\},\ m \geq n$$
$$\Gamma \vdash < type_{1,i} > \leq < type_{2,i} > \quad \text{for each } < name_{1,i} >=< name_{2,i} >$$
$$\Rightarrow \Gamma \vdash \{| < name_{1,1} >:< type_{1,1} >, \ldots, < name_{1,m} >:< type_{1,m} > |\}$$
$$\leq \{| < name_{2,1} >:< type_{2,1} >, \ldots, < name_{2,n} >:< type_{2,n} > |\}$$

3. The object-oriented inheritance

A class (child class) inherits state and behavior from its superclass (parent class). Inheritance provides a powerful and natural mechanism for organizing and structuring software programs. The instance variables and the methods of the child class are an extension of the structure and the behavior of the parent class. The child class extends the properties, the structure of the parent class, and constitutes a limitation of the meaning.

Definition 3.1 (The principle of substitutability)**.** *An instance of the child class can imitate the behavior of a parent class and can not be distinguished from an instance of the parent class if it is used in a similar situation. If C is the child oclass and P is the parent class, $C = subst(P)$ means that each instance of C may be used instead of an instance of P.*

Definition 3.2 (Subtype)**.** *A subtype is a class which fulfills the principle of substitutability ($C = subst(P)$).*

Informally, a type σ is a subtype of τ, written $\sigma \leq \tau$, if an expression of type σ can be used in any context that expects an expression of type τ.

The rule for subtyping functions states that $\sigma \to \tau \leq \sigma' \to \tau'$ iff $\sigma' \leq \sigma$ and $\tau \leq \tau'$.

This is formalized by extending our λ-calculus with a subtype relation, written $\Gamma \vdash S \leq T$ to mean that S is a subtype of T under assumptions Γ.

Definition 3.3 (Subclass)**.** *A subclass is an arbitrary class created by inheritance, regardless of the principle of substitutability ($C \neq subst(P)$).*

A subclass may also override the definitions of methods it would otherwise inherit by redefining them. Because a subclass inherits code for methods, it also inherits interface type information for the methods that it does not override.

We write $\Gamma \vdash S < T$ to mean that S is not a subtype of T, but is a subclass of T under assumptions Γ.

In the object-oriented paradigm a class is a prototype that defines the variables and the methods common to all objects of a certain kind.

Let $v_1, v_2, \ldots, v_i$ be the variables and let $m_1, m_2, \ldots, m_j$ be the methods, let $V = \{V_1, V_2, \ldots, V_i\}$ be the typeclass of each variable and let $M = \{M_1, M_2, \ldots, M_j\}$ be the typeclass of each method (the signature of methods) for a given class S, where i is the number of variables and j is the number of methods for a class $S = V \cup M$.

In that case, the subtyping rule between classes S and T is:

$$V = \{V_1, V_2, \ldots, V_i\}, \quad M = \{M_1, M_2, \ldots, M_j\},$$

$$V' = \{V'_1, V'_2, \ldots, V'_{i'}\}, \quad M' = \{M'_1, M'_2, \ldots, M'_{j'}\},$$

$$V \subseteq V', \ M \subseteq M', \ i \geq i', \ j \geq j',$$

$$S = V \cup M, \ T = V' \cup M',$$

$$T = \text{subst}(S),$$

$$\Gamma \vdash \{|\ v_1 : V_1, \ldots, v_i : V_i, m_1 : M_1, \ldots, m_j : M_j\ |\} \in *,$$

$$\Gamma \vdash \{|\ v'_1 : V'_1, \ldots, v'_{i'} : V'_{i'}, m'_1 : M'_1, \ldots, m'_{j'} : M'_{j'}\ |\} \in *$$

$$\Rightarrow \Gamma \vdash S \leq T$$

For a better formalization of $T = \text{subst}(S)$ or $T \neq \text{subst}(S)$ we must give the subtyping (or subclassing) rule for each kind of inheritance.

3.1. **Specialization.**

Definition 3.4 (Specialization)**.** *The child class is a specialized form of the parent class.*

Additional functionalities, principle of substitutability is guaranteed. Does not change the old variables and methods (from parent class), but brings in new variables and new methods in child class.

$$V = \{V_1', V_2', \ldots, V_{i'}', \ldots, V_i\}, \quad M = \{M_1', M_2', \ldots, M_{j'}', \ldots, M_j\},$$

$$V' = \{V_1', V_2', \ldots, V_{i'}'\}, \quad M' = \{M_1', M_2', \ldots, M_{j'}'\},$$

$$V \subseteq V',\ M \subseteq M',\ i > i',\ j > j',$$

$$S = V \cup M,\ T = V' \cup M',$$

$$\Gamma \vdash \{|\ v_1 : V_1, \ldots, v_i : V_i, m_1 : M_1, \ldots, m_j : M_j\ |\} \in *,$$

$$\Gamma \vdash \{|\ v_1' : V_1', \ldots, v_{i'}' : V_{i'}', m_1' : M_1', \ldots, m_{j'}' : M_{j'}'\ |\} \in *$$

$$\Rightarrow \Gamma \vdash S \leq T$$

3.2. **Specification.**

Definition 3.5 (Specification)**.** *The parent class defines an interface, the child class gives the implementation.*

Changes the old variables of parent class, no additional functionalities, principle of substitutability is guaranteed.

$$V = \{V_1, V_2, \ldots, V_i\}, \quad M = \{M_1', M_2', \ldots, M_j'\},$$

$$V' = \{V_1', V_2', \ldots, V_i'\}, \quad M' = \{M_1', M_2', \ldots, M_j'\},$$

$$V \subseteq V',\ M = M',$$

$$S = V \cup M,\ T = V' \cup M',$$

$$\Gamma \vdash \{|\ v_1 : V_1, \ldots, v_i : V_i, m_1 : M_1', \ldots, m_j : M_j'\ |\} \in *,$$

$$\Gamma \vdash \{|\ v_1' : V_1', \ldots, v_i' : V_i', m_1' : M_1', \ldots, m_j' : M_j'\ |\} \in *$$

$$\Rightarrow \Gamma \vdash S \leq T$$

3.3. **Construction.**

Definition 3.6 (Construction)**.** *The parent class provides the functionality, but gives no logical context to the child class.*

Typical kind of subclassing, changes the old variables, methodes, adds new variables and methods, no substitutability.

$$V = \{V_1, V_2, \ldots, V_i\}, \quad M = \{M_1, M_2, \ldots, M_j\},$$
$$V' = \{V_1', V_2', \ldots, V_{i'}'\}, \quad M' = \{M_1', M_2', \ldots, M_{j'}'\},$$
$$V \subseteq V', \ M \subseteq M', \ i \geq i', \ j \geq j',$$
$$S = V \cup M, \ T = V' \cup M',$$
$$\Gamma \vdash \{| \ v_1 : V_1, \ldots, v_i : V_i, m_1 : M_1, \ldots, m_j : M_j \ |\} \in *,$$
$$\Gamma \vdash \{| \ v_1' : V_1', \ldots, v_{i'}' : V_{i'}', m_1' : M_1', \ldots, m_{j'}' : M_{j'}' \ |\} \in *$$
$$\Rightarrow \Gamma \vdash S < T$$

3.4. **Generalization.**

Definition 3.7 (Generalization)**.** *The child class extends the functionality of the parent class, creates more general instances.*

Like the construction, but we can tell absolutly nothing about substitutability. It depends on particular cases.

$$V = \{V_1, V_2, \ldots, V_i\}, \quad M = \{M_1, M_2, \ldots, M_j\},$$
$$V' = \{V_1', V_2', \ldots, V_{i'}'\}, \quad M' = \{M_1', M_2', \ldots, M_{j'}'\},$$
$$V \subseteq V', \ M \subseteq M', \ i \geq i', \ j \geq j',$$
$$S = V \cup M, \ T = V' \cup M',$$
$$\Gamma \vdash \{| \ v_1 : V_1, \ldots, v_i : V_i, m_1 : M_1, \ldots, m_j : M_j \ |\} \in *,$$
$$\Gamma \vdash \{| \ v_1' : V_1', \ldots, v_{i'}' : V_{i'}', m_1' : M_1', \ldots, m_{j'}' : M_{j'}' \ |\} \in *$$
$$\Rightarrow \Gamma \vdash S \leq ? < T$$

3.5. **Extension.**

Definition 3.8 (Extension)**.** *The child class adds further functionalities to the parent class, but does not change any inherited behavior.*

The principle of substitutability is guaranted.

$$V = \{V_1', V_2', \ldots, V_i'\}, \quad M = \{M_1, M_2, \ldots, M_j\},$$
$$V' = \{V_1', V_2', \ldots, V_i'\}, \quad M' = \{M_1', M_2', \ldots, M_j'\},$$
$$V = V', \ M \subseteq M',$$
$$S = V \cup M, \ T = V' \cup M',$$
$$\Gamma \vdash \{| \ v_1 : V_1', \ldots, v_i : V_i', m_1 : M_1, \ldots, m_j : M_j \ |\} \in *,$$
$$\Gamma \vdash \{| \ v_1' : V_1', \ldots, v_{i'}' : V_{i'}', m_1' : M_1', \ldots, m_{j'}' : M_j' \ |\} \in *$$
$$\Rightarrow \Gamma \vdash S \leq T$$

3.6. **Limitation.**

Definition 3.9 (Limitation)**.** *The child class restricts the use of some of the behaviors inherited from the parent class.*

The principle of substitutability is not guaranted.

$$V = \{V'_1, V'_2, \ldots, V'_i\}, \quad M = \{M_1, M_2, \ldots, M_j\},$$
$$V' = \{V'_1, V'_2, \ldots, V'_i\}, \quad M' = \{M'_1, M'_2, \ldots, M'_j\},$$
$$V = V', \ M \subseteq M',$$
$$S = V \cup M, \ T = V' \cup M',$$
$$\Gamma \vdash \{|\ v_1 : V'_1, \ldots, v_i : V'_i, m_1 : M_1, \ldots, m_j : M_j\ |\} \in *,$$
$$\Gamma \vdash \{|\ v'_1 : V'_1, \ldots, v'_i : V'_i, m'_1 : M'_1, \ldots, m'_j : M'_j\ |\} \in *$$
$$\Rightarrow \Gamma \vdash S < T$$

3.7. **Variance.**

Definition 3.10 (Variance)**.** *The parent and the child class are variant of each other, the class – subclass relationship is arbitrary.*

The principle of substitutability is not guaranted.

$$V = \{V_1, V_2, \ldots, V_i\}, \quad M = \{M_1, M_2, \ldots, M_j\},$$
$$V' = \{V'_1, V'_2, \ldots, V'_i\}, \quad M' = \{M'_1, M'_2, \ldots, M'_j\},$$
$$V \subseteq V', \ M \subseteq M',$$
$$S = V \cup M, \ T = V' \cup M',$$
$$\Gamma \vdash \{|\ v_1 : V_1, \ldots, v_i : V_i, m_1 : M_1, \ldots, m_j : M_j\ |\} \in *,$$
$$\Gamma \vdash \{|\ v'_1 : V'_1, \ldots, v'_i : V'_i, m'_1 : M'_1, \ldots, m'_j : M'_j\ |\} \in *$$
$$\Rightarrow \Gamma \vdash S < T$$

3.8. **Combination.**

Definition 3.11 (Combination)**.** *The child class inherits features from two or more parent classes. It is commonly called multiple inheritance.*

Changes the old variables, methods, brings in new variables, methods. We can tell absolutly nothing about substitutability. It depends on particular cases.

$$V = \{V_1, V_2, \ldots, V_i\}, \quad M = \{M_1, M_2, \ldots, M_j\},$$
$$V' = \{V'_1, V'_2, \ldots, V'_{i'}\}, \quad M' = \{M'_1, M'_2, \ldots, M'_{j'}\},$$
$$V \subseteq V', \ M \subseteq M', \ i \geq i', \ j \geq j',$$
$$S = V \cup M, \ T = V' \cup M',$$
$$\Gamma \vdash \{|\ v_1 : V_1, \ldots, v_i : V_i, m_1 : M_1, \ldots, m_j : M_j\ |\} \in *,$$
$$\Gamma \vdash \{|\ v'_1 : V'_1, \ldots, v'_{i'} : V'_{i'}, m'_1 : M'_1, \ldots, m'_{j'} : M'_{j'}\ |\} \in *$$
$$\Rightarrow \Gamma \vdash S \leq ? < T$$

4. Summary - Inheritance

The following table contains a general view of object-oriented inheritance kinds:

Kind	*New variables*	*New methods*	*Substi-tutability*	*Changes old variables*	*Changes old methods*
specialization	Yes	Yes	Yes	No	No
specification	No	No	Yes	Yes	No
construction	Yes	Yes	No	Yes	Yes
generalization	Yes	Yes	?(Yes,No)	Yes	Yes
extension	No	Yes	Yes	No	No
limitation	No	No	No	No	Yes
variance	No	No	No	Yes	Yes
combination	Yes	Yes	?(Yes,No)	Yes	Yes

5. The object-oriented polymorphism

According to the object-oriented polymorphism, one message (the same message) can be different interpreted by the objects (Methods with the same names, signatures and with different bodies). Polymorphism is a natural characteristic of object-oriented languages based on the principles of message passing, inheritance and substitutability [6].

For a better formalization we must give the subtyping rule for each kind of polymorphism.

5.1. **Overloading.**

Definition 5.1 (Overloading)**.** *A function name denotes more than one possible statement sequence.*

Overloading extends the syntax of the programming language. For example, overloading the '+' operator for *Complex* numbers (*Complex* is a class). Subtyping rule: see Overriding.

5.2. **Polymorphic parameters.**

Definition 5.2 (Polymorphic parameters)**.** *One method (function or procedure) can be called with different type of arguments.*

A method can be redeclared with a different parameter signature from its ancestor, it overloads the inherited method without hiding it. Calling the method in a descendant class activates whichever implementation matches the parameters in the call.

$$V = \{V_1, V_2, \ldots, V_i\}, \quad M = \{M_1, M_2, \ldots, M_j\},$$

$$V' = \{V'_1, V'_2, \ldots, V'_{i'}\}, \quad M' = \{M'_1, M'_2, \ldots, M'_{j'}\},$$
$$V \subseteq V',\ M \subseteq M',\ i \geq i',\ j \geq j',$$
$$S = V \cup M,\ T = V' \cup M',$$
$$\Gamma \vdash \{|\ v_1 : V_1, \ldots, v_i : V_i, m_1 : M_1, \ldots, m_j : M_j\ |\} \in *,$$
$$\Gamma \vdash \{|\ v'_1 : V'_1, \ldots, v'_{i'} : V'_{i'}, m'_1 : M'_1, \ldots, m'_{j'} : M'_{j'}\ |\} \in *$$
$$\exists k \in \{1, \ldots, j\} : name(m_k) = name(m'_k) \wedge signature(m_k) \neq signature(m'_k)$$
$$\Rightarrow \Gamma \vdash S \leq T$$

Where *name(m)* is the name of the method, and *signature(m)* is the signature (name and parameter list) of the method.

5.3. **Deferring.**

Definition 5.3 (Deffering)**.** *The parent class declares only the method, the child class implements it.*

Commonly called abstract polymorphism.

$$V = \{V_1, V_2, \ldots, V_i\}, \quad M = \{M_1, M_2, \ldots, M_j\},$$

- no method bodies in M, just the declarations.

$$V' = \{V'_1, V'_2, \ldots, V'_{i'}\}, \quad M' = \{M'_1, M'_2, \ldots, M'_{j'}\},$$
$$V \subseteq V',\ M \subseteq M',\ i \geq i',\ j \geq j',$$
$$S = V \cup M,\ T = V' \cup M',$$
$$\Gamma \vdash \{|\ v_1 : V_1, \ldots, v_i : V_i, m_1 : M_1, \ldots, m_j : M_j\ |\} \in *,$$
$$\Gamma \vdash \{|\ v'_1 : V'_1, \ldots, v'_{i'} : V'_{i'}, m'_1 : M'_1, \ldots, m'_{j'} : M'_{j'}\ |\} \in *$$
$$\forall k \in \{1, \ldots, j\} : signature(m_k) = signature(m'_k)$$
$$\Rightarrow \Gamma \vdash S \leq T$$

5.4. **Overriding.**

Definition 5.4 (Overriding)**.** *A child class changes the meaning of a method, which was defined in the parent class.*

Overriding a method means extending or refining it, rather than replacing it. A descendant class can override any of its inherited virtual methods. It is the common and the most used case of polymorphism.

$$V = \{V_1, V_2, \ldots, V_i\}, \quad M = \{M_1, M_2, \ldots, M_j\},$$
$$V' = \{V'_1, V'_2, \ldots, V'_{i'}\}, \quad M' = \{M'_1, M'_2, \ldots, M'_{j'}\},$$
$$V \subseteq V',\ M \subseteq M',\ i \geq i',\ j \geq j',$$
$$S = V \cup M,\ T = V' \cup M',$$
$$\Gamma \vdash \{|\ v_1 : V_1, \ldots, v_i : V_i, m_1 : M_1, \ldots, m_j : M_j\ |\} \in *,$$
$$\Gamma \vdash \{|\ v'_1 : V'_1, \ldots, v'_{i'} : V'_{i'}, m'_1 : M'_1, \ldots, m'_{j'} : M'_{j'}\ |\} \in *$$

$$\exists k \in \{1, \ldots, j\} : signature(m_k) = signature(m'_k)$$
$$\Rightarrow \Gamma \vdash S \leq T$$

6. Summary - Polymorphism

The following table contains a general view of object-oriented polymorphism kinds:

Kind	*Same names*	*Same bodies*	*Same signatures*	*Abstract methods*	*Extends the syntax*
overloading	Yes	No	Yes	No	Yes
parameters	Yes	No	No	No	No
deferring	Yes	No	Yes	Yes	No
overriding	Yes	No	Yes	No	No

References

[1] Atsuchi Igarashi, Benjamin C. Pierce, *Foundations for Virtual Types*, ECOOP'99, LNCS 1628, 1999, 161–185.
[2] Martin Abadi, Luca Cardelli, *A Theory of Objects*, Springer-Verlag, 1996.
[3] Luca Cardelli, *Notes about* $F^{\omega}_{\leq}$, http://citeseer.nj.nec.com/cs, Unpublished manuscript, October 1990
[4] Kathleen Fisher, John C. Mitchell, *The Development of Type Systems for Object-Oriented Languages*, Stanford University, STAN-CS-TN-96-30
[5] Benjamin C. Pierce, *Type Systems*, Draft, 2000.
[6] Luca Cardelli, Peter Wagner, *On Understanding Types, Data Abstraction and Polymorphism*, ACM Computing Surveys, 17 (4), 1995, 471–522

Department of Computer Systems, Babeş-Bolyai University, Cluj-Napoca
E-mail address: klehel@cs.ubbcluj.ro

John von Neumann Faculty of Informatics, Budapest Polytechnic, Budapest
E-mail address: legradi@nik.bmf.hu

Department of General Computer Science, Eötvös Loránd University, Budapest
E-mail address: csz@inf.elte.hu