STUDIA UNIV. "BABEŞ-BOLYAI", INFORMATICA, Volume I, Number 2, October 1996

HOW TO MODEL MESSAGE PASSING BY LAMBDA-CALCULUS

S. MOTOGNA

Abstract. The paper contains some aspects concerning message passing in object oriented languages which are specified through lambda- calculus. This aspect is most often not taken in view while other object oriented features are well presented. The semantics used is based on lambda calculus with types. In this paper we prove that the two models for message passing are, in essence, equivalent.

1. Introduction

Object oriented programming has established itself as the most modern and natural way of software development. That's why recent studies try to formalise the OOP characteristics as better as possible. One of the way in which the research is carried on is lambda calculus, most of the studies following Cardelli and Wegner point of view [3].

The notions with which the OOP deals are objects, classes and methods (or messages, as method calls). The main features of OOP are: inheritance (or subtyping), polymorphism and information hiding (followed up from data abstraction). Various extensions of typed lambda calculus model these features.

In my point of view message itself is not "a big deal". Message passing seems to me a more important aspect, a more suitable notion, because is the only way in which the object can communicate.

A message syntax can be supposed of the following form:

message.receiver

without restraining its generality, because all object oriented languages has a similar form (if the receiver is not specified it is implicit).

The transmission of a message implies the existence of a receiver, which is an object (or more generally an expression returning an object); when it receives a message, the system, at run-time, selects among the methods defined for that

1991 *Mathematics Subject Classification.* 68NO5, 68Q55.
1991 *CR Categories and Descriptors.* D.1.5 [Programming Techniques] Object oriented programming, F.4.1 [Mathematical logic and formal languages] Mathematical logic, Lambda-calculus and related systems.

object the one whose name corresponds to the message name, if this exists. The existence of such a method should be checked at compile-time.

2. Message passing

There are two ways to understand message passing:

a): the first one is based on Cardelli's point of view [2] and consists of: the object is considered a record in which methods are its fields and whose labels are the messages of the corresponding methods. This point of view is know as "objects as records" analogy and can be found in [3], [5], [7].

b): the second way is used in the language CLOS and consists of: in the context of typed functional languages, message passing is viewed as a functional application in which the message is the function (identifier of the function) and the receiver is its argument [4].

In some sense these two interpretations are similar because records can be represented as functions from labels (messages) to values.

2.1. "Objects as records".

Let's review some aspects in favour of this choice:

- in Simula (considered an ancestor language in object oriented paradigm) objects are records with possibly functional components
- a record selection usually requires the selection label to be known at compile-time (allowing compile-time checking of the called method).

In [2] is presented how records can model the basic features of objects, including: inheritance, multiple inheritance, message passing, private instance variables and the special variable "self". The method chosen is based on typed lambda calculus with records and variants.

The subtype relation is defined as follows [2]: a record type γ is a subtype (written $\leq$) of a record type γ' if γ has all the fields of γ', and possibly more, and the common fields of γ and γ' are in the $\leq$ relation. The typing rules are:

[Rule 1a:] if $e_1 : \gamma_1$ and ... and $e_n : \gamma_n$ then $(a_1 = e_1, \ldots, a_n = e_n) : (a_1 : \gamma_1, \ldots, a_n : \gamma_n)$

[Rule 2a:] $\iota \leq \iota$ (ι a basic type, like *int* and *bool*)

$$\gamma_1 \leq \gamma_1', \ldots, \gamma_n \gamma_n' \Rightarrow (a_1 : \gamma_1, \ldots, a_{n+m} : \gamma_{n+m}) \leq (a_1 : \gamma_1', \ldots, a_n : \gamma_n')$$

[Rule 3a:]if $a : \gamma$ and $\gamma \leq \gamma'$ then $a : \gamma'$

[Rule 4a:]if $f : \sigma \to \gamma$ and $a : \sigma$ then $f(a)$ is meaningful, and $f(a) : \gamma$

[Rule 5a:] if $f : \sigma \to \gamma$ and $a : \gamma'$, where $\sigma' \leq \sigma$ then $f(a)$ is meaningful, and $f(a) : \gamma$

[Rule 6a:] if $\sigma' \leq \sigma$ and $\gamma \leq \gamma'$ then $\sigma \to \gamma \leq \sigma' \to \gamma'$

2.2. **"Message passing as functional application"**. We note first that we must distinguish methods from functions, at least two main aspects being considered:

- *overloading* (the answer of two objects to the same message can differ, depending on the type of the object). Thus, messages are identifiers of the overloaded functions and in message passing the first argument of the overloaded function is represented by the receiver and the one on whose type the selection of the code to be executed is based. Each method constitutes a branch of the overloaded function referred by the message it is associated to;
- *late binding*: The difference consists in the fact that a function is bind to its meaning at compile-time, while the meaning of a method can be decided only at run-time when the actual type of the receiver is known. This feature is called late binding and it's shown up in the combination between overloading and subtyping.

Combining overloading with late binding implies a new distinction between message passing and ordinary functions. Overloading and late binding requires a restriction in the evaluation technique of arguments: while ordinary function application can be dealt with by either call-by-value or call-by-name, overloaded application with late binding can be evaluated only when the run-time type of the argument is known, i.e. when the argument is fully evaluated (closed and in normal form). In view of our analogy "messages as overloaded functions" this corresponds to say that message passing acts by call-by-value or, more generally, only closed and normal terms responds to messages.

An overloaded function is formed by a set of ordinary functions (lambda-abstractions), each one forming a different branch, and the notation is:

$$(M\&N)$$

for an overloaded function with two branches M and N that will be selected according to the type of the argument.

The subtyping relation is defined as follows:

$$\text{if } U_2 \leq U_1 \text{ and } V_1 \leq V_2 \text{ then } U_1 \to V_1 \leq U_2 \to V_2$$

and

$$\begin{array}{l}\text{if } \forall i \in I, \exists j \in J \text{ such that } U'_j -> V'_j \leq U''_i \to V''_i \\ \textit{then} \{U'_j \to V'_j\}_{j\in J} \leq \{U''_i \to V''_i\}_{i\in I}.\end{array}$$

The type-checking rules are [4]:

[Rule 1b:] $x^V : V$ x^V denotes $x : V$
[Rule 2b:] if $\lambda x^U.M : U \to V$ then $M : V$
[Rule 3b:] if $M : U \to V$ and $N : W \leq U$ then $MN : V$
[Rule 4b:] $\in: \{\}$

[Rule 5b:] if $M : W_1 \leq \{U_i \to V_i\}_{i\leq(n-1)}$ and $N : W_2 \leq U_n \to V_n$ then $(M\&^{\{Ui->Vi\}i\leq n}N) : \{U_i \to V_i\}_{i\leq n}$

[Rule 6b:] if $M : \{U_i \to V_i\}_{i\in I}$ and $N : U$, $U_j = \min_{i\in I}\{U_i | U \leq U_i\}$ then $M * N : V_j$ $M * N$ denotes the overloaded application

3. Comparative study

First of all we must note that the same notation has been used for typing and subtyping relations.

If we study the two sets of typing rules we must find an "equivalence" between them. We will follow the equivalence from the first set of rules (from 2.a) to the second set of rules (from 2.b), since records can be represented as functions from labels (messages) to values.

[Rule 1a]: e_i are values of the corresponding type γ_i (for $1 \leq i \leq n$). Then if we proceed with labelling in a record we obtain Rule 1a. In other words the type of a record is obtained from the types of the labels. Since it is a rule specialized for records we can't find a match in the second set of rules.

[Rule 2a] We can decompose this rule as follows:

$$\begin{array}{l}\text{if } \gamma_1 \leq \gamma_1' \vdots \gamma_n \leq \gamma_n' \text{ then} \\ (a_1 : \gamma_1, \ldots, a_n : \gamma_n) \leq (a_1 : \gamma_1', \ldots, a_n : \gamma_n') \\ (a_1 : \gamma_1, \ldots, a_{n+m} : \gamma_{n+m}) \leq (a_1 : \gamma_1, \ldots, a_n : \gamma_n)\end{array}$$

Comparing this rule with [Rule 5b] which states that overloading a function (with n+1 members) with a new member N which type is a subtype of $U_n \to V_n$ the result has the type $\{U_i \to V_i\}_{i\leq n}$.

We can generalize [Rule 5b] as follows:

$$\begin{array}{l}\text{if } M : W_1 \leq \{U_i \to V_i\}_{i\leq(n-1)} \text{ and } N_1 : W_{21}, \ldots, N_m : W_{2m} \\ \text{then } (M\&^{\{Ui->Vi\}i\leq n}N_1\&\vdots\&N_m) : \{U_i \to V_i\}_{i\leq n+m}\end{array}$$

and the correspondence with [Rule 2a] is observable.

[Rule 3a] is in fact the principle of subtyping.

The following 3 rules have as basic studying object the typing rules when functions are involved.

[Rule 4a] states that the type of the function value can be deducted from the definiton of the function, if it is meaningful. If we describe the function definition in the lambda-calculus notation we obtain the operation called application. And [Rule 2b] it is the exact transcription of the [Rule 4a].

[Rule 5a] is concerned with the case when we consider a subtype of the domain type of the function. [Rule 3b] states a similar situation when abstraction is involved.

[Rule 6a] is similar to the definition of the subtyping relation from 2b.

4. Conclusions

When necessary, we develope some theory from the most suitable point of view. We must never forget that this theory has to be consistent. Both ways of modelling message passing meet this criterion.

On the other hand, it might be possible that these two theories have to be unified. Then the problem of equivalence is asked.

We have argued that in principal the typing rules can be deduced one from another. The study takes into account only one direction of this deduction (from "objects as records" study to "message passing as functional application" study). The reverse sense of the equivalence can be deducted in a similar way.

Finally, we will note once again that the main idea of this deduction is the fact that records can be represented as functions from labels (messages) to values.

References

[1] K.B. Bruce, G. Longo, *A modest model of records, inheritance and bounded quantification*, Information and Computation, 87(1/2), 1990, pg. 196-240.

[2] L. Cardelli, *A semantics of multiple inheritance*, Semantics of Data Types, Lectures Notes in Computer Science, 173, Springer Verlag, 1984, pg. 51-67.

[3] L. Cardelli, P. Wegner, *On understanding types, data abstraction and polymorphism*, Computing Surveys, 17(4), dec. 1995, pg. 471-522.

[4] G. Castagna, G. Ghelli, G. Longo, *A semantics for λ-early: a calculus with overloading and early binding*, ACM Conference on Lisp and functional languages, 1994, pg. 107- 123.

[5] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall International Series, 1988.

[6] S. Motogna, *Formal Specification for Smalltalk through Lambda-Calculus. A Comparative Study*, Studia 3/1993.

[7] S. Motogna, V. Prejmerean, *Various kinds of inheritance*, Studia 3/1992, pg. 75-80

Babeş-Bolyai University, Faculty of Mathematics and Informatics,
RO 3400 Cluj-Napoca, str. Kogălniceanu 1, Romania
E-mail address: motogna@cs.ubbcluj.ro