JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2005

Vol. 4, No. 2, March–April 2005

Accessing Objects Locally in Object-Oriented Languages

Keehang Kwon, Dept. of Computer Eng., DongA University, South Korea

We propose method invocation constructs that allow objects to be accessed *locally* in object-oriented languages. The major construct is the expression of the form $O.E$ where O is an object and E is an expression. This construct has the following operational semantics: add the object O to the program context in the course of evaluating E. Thus, the object O is available only in the course of evaluating E. Consequently, the program context consists of only the currently active objects and thus is managed in a memory-efficient way. Finally we compare this notion with cache systems.

1 INTRODUCTION

Most object-oriented languages[AC96, Meyer] – Java, C#, *etc*– lack devices for accessing objects locally when they invoke a method. Lacking such a device, it is cumbersome for a programmer to specify the exact unloading times of an object from the program context during execution. Consequently, programs are executed as a monolithic collection of objects in this context.

We consider an example to illustrate this aspect. Let us assume that the objects A_i, for $1 \leq i \leq 8$, are defined as below.

$$\begin{array}{l} \textit{object } A_i. \\ \textit{var } \_w. \\ a(X) = A_{i+1}.a(X). \\ \vdots \\ \textit{end } A_i \end{array}$$

The object A_9 is defined as below with its field $\_w$ being initialized to 9.

$$\begin{array}{l} \textit{object } A_9 \\ \textit{var } \_w = 9. \\ a(X) = X * X \\ b(X) = X + \_w. \\ \textit{end } A_9 \end{array}$$

Now the attempt to evaluate the expression $A_1.a(2)+A_9.b(3)$ proceeds as follows: it first loads the objects $A_1, \ldots, A_9$ into the program context, evaluates the first argument $A_1.a(2)$, evaluates the second argument $A_9.b(3)$ and then add the two results, all from the same program context. The evaluation steps for the first and second arguments are shown below.

$$\begin{array}{lll}
A_1, \ldots, A_9 & ?- & A_1.a(2) \quad \% \text{ first argument} \\
 & \vdots & \\
A_1, \ldots, A_9 & ?- & A_9.a(2) \\
A_1, \ldots, A_9 & ?- & 2 * 2 \\
 & & \\
A_1, \ldots, A_9 & ?- & A_9.b(3) \qquad \% \text{ second argument} \\
A_1, \ldots, A_9 & ?- & 3 + A_9.\_w \\
A_1, \ldots, A_9 & ?- & 3 + 9
\end{array}$$

In the computation above, there are much redundancies in the program context. For instance, none of the objects $A_1, \ldots, A_8$ are needed in the program context when the second argument $A_9.b(3)$ is evaluated.

This paper introduces constructs that cope with this redundancy. The major construct is the expression of the form $O.E$ where O is an object and E is an expression. This one has the following intended semantics: the object O is intended to be added to the program context in the course of evaluating E. Hence, the object O will be unloaded from the program context after the evaluation of E is done. This expression thus supports the idea of accessing objects locally. Another construct is of the form $(private\ x\backslash\ O).E$ and has the following intended semantics: the variable x in O is intended to be replaced with a new name before evaluating $O.E$. This expression thus supports the idea of private constants.[1]

In this paper we present the syntax and semantics of this extended language, and show some examples of its use. The remainder of this paper is structured as follows. In the next section, we describe a modification to the method invocation expression. In Section 3, we present some examples. Section 4 concludes this paper. In particular, we discuss how the notion of accessing objects locally makes cache replacement algorithms redundant.

2 THE LANGUAGE

Most object-oriented languages use the qualified expression $(o.f)(a)$, which stands for method invocation of f in an object o with argument a. The main drawback of this expression is that the object o will remain in the program context even after $f(a)$ is evaluated.

In this paper, we propose a new construct for method invocation: namely $o.(f(a))$

[1]These two constructs are motivated from Miller's work in the logic programming paradigm [Mil89b, Mil89a].

instead of $(o.f)(a)$. Its meaning is the following: add o to the program context in the course of evaluating $f(a)$. This construct naturally incorporates a notion of locality for the object o.[2] This new construct also applies to a field name. The notation $o.\_w$ means the following: add o to the program context in the course of evaluating $\_w$. In the sequel, we assume that field names are prefixed by an underline for the reason of convenience.

The object calculi to be considered is described by E- and O-formulas given by the syntax rules below:

$$E ::= c \mid x \mid \_w \mid f(E, \dots, E) \mid O.E$$

$$O ::= \text{an object} \mid \textit{private } x \backslash\ O$$

In the rules above, c, x, $\_w$ respectively represents a constant, a variable and a field name. In the object-calculi to be considered, E-formulas are expressions and a list of O-formulas will constitute programs.

The notion of evaluating an expression is presented below. In describing the idea of an evaluation, we write $[t_1/x_1, \dots, t_n/x_n]e$ to denote the application of a substitution $\{\langle x_i, t_i\rangle | 1 \leq i \leq n\}$ to a term e.

The rules for evaluating expressions in our language are based on "call-by-value" mechanism in the sense that the arguments of a function are evaluated first. In describing the rules, it is assumed that E cannot be a variable as all variables should have been bound to a value before being evaluated.

Definition 1 *Let E be an expression and let $\mathcal{P}$ be a list of objects. Then the notion of evaluating $\langle \mathcal{P}, E\rangle$ is defined as follows:*

(1) If E is a constant c, then c.

(2) If E is a field name $\_w$ and o is the most recently added object that has a field name $\_w$, then it is the value of $\_w$ in o.

(3) If E is $f(E_1, \dots, E_n)$, r_i is the value of evaluating $\langle \mathcal{P}, E_i\rangle$, and the newest method f defined in $\mathcal{P}$ is of the form $f(x_1, \dots, x_n) = E_1$, then evaluate $\langle \mathcal{P}, [r_1/x_1, \dots, r_n/x_n]E_1\rangle$.

(4) If E is $O.E_1$ and O is private $x_1 \backslash \dots$ private $x_n \backslash\ O_1$, then evaluate $\langle [a_1/x_1, \dots, a_n/x_n]O_1 :: \mathcal{P}, E_1\rangle$ where each a_i is a new name and :: is the list constructor.

In the above rules, the symbols . and *private* $x\backslash$ provide scoping mechanisms: they allow, respectively, for the augmentation of the program and the introduction

[2]If f is defined in o, this new definition will override the old one (if any) in the program context.

of new names in the course of evaluating an expression. The *private* construct in O-formulas provides a means for information hiding. Notice that method invocation of the form $o_1.(o_2.\ldots(o_n.(f(a))\ldots)$ is also allowed. This expression means that it will add $o_1, o_2, \ldots$ and finally o_n to the program context in the course of evaluating $f(a)$. This expression provides a form of dynamic inheritance. For instance, $o_1.(o_2.(f(a)))$ allows that o_2 inherits o_1 in the course of evaluating$f(a)$.

3 EXAMPLES

We reconsider the example in Section 1 to illustrate the dynamic aspect of the new method invocation construct. Let us assume that the objects A_i, for $1 \leq i \leq 9$, are defined as before.

The attempt to evaluate the expression $A_1.a(2) + A_9.b(3)$ proceeds as follows: it evaluates the first argument $A_1.a(2)$, evaluates the second argument $A_9.b(3)$, and then add the two results, all from the empty program context. The evaluation steps for the first argument are shown below. The initial context is empty, but dealing with $A_1.a(2)$ causes A_1 to be added to it. The expression to be evaluated now is $a(2)$. There is only one definition for a in the context and $a(2)$ is replaced with $A_2.a(2)$. The object A_2 is therefore added to the program context and the expression to be evaluated reduces to $a(2)$. There are two definitions for a in the context and the one in A_2 is used as the newest definition overrides the previous ones. Continuing with the evaluation attempt, it eventually computes the result which is $2 * 2$.

$$\begin{array}{rcll}
 & ?- & A_1.a(2) & \%\text{ evaluating first argument} \\
A_1 & ?- & a(2) & \%\ A_1\text{ is loaded into program context} \\
A_1 & ?- & A_2.a(2) & \\
A_2, A_1 & ?- & a(2) & \\
A_2, A_1 & ?- & A_3.a(2) & \\
\vdots & & & \\
A_9, \ldots, A_1 & ?- & a(2) & \\
A_9, \ldots, A_1 & ?- & 2 * 2 &
\end{array}$$

The evaluation steps for the second argument are similar as shown below. Note that the initial context is empty, but dealing with $A_9.b(3)$ causes A_9 to be added to it. The expression to be evaluated now is $b(3)$. There is only one definition for b in the context and $b(3)$ is replaced with $3 + \_w$. There are a field name $\_w$ in A_9 in the context and its initial value is 9. It eventually computes the result which is $3 + 9$.

$$\begin{array}{rcll}
 & ?- & A_9.b(3) & \%\text{ evaluating second argument} \\
A_9 & ?- & b(3) & \\
A_9 & ?- & 3 + \_w & \\
A_9 & ?- & 3 + 9 &
\end{array}$$

In the computation above, it is easily observed that there are no redundancies in the program context. For instance, the computation uses only A_9 to evaluate $A_9.b(3)$.

The constuct considered causes objects to be added dynamically to the program. Expressions must therefore be evaluated relative to particular program contexts. Regarding implementation, this is not a practical scheme and can be improved upon by noting that program contexts change in a stack-disciplined fashion. Thus program contexts can be implemented using the program stack which permits the incremental addition and subsequent retraction of objects.

Our language permits method names to be made private to an object using the *private* construct. This allows for the hiding of a method name in the object. The names of the method names listed then become unavailable outside the object. An example of the use of this construct is provided by an expression of the form $(private\ b\backslash\ A_9).a(3)$. This has the effect of adding A_9 — after replacing b with a new name — to the program before evaluating $a(3)$.

4 CONCLUSION

We have examined a new construct for method invocation in object-oriented languages. The program context in our execution model maintains only the active parts of the program. Similarly the notion of a cache tries to maintain the active parts of the program. Hence it is interesting to compare our execution model with the model that employs the notion of cache.

In both models, objects enter the program context "on demand". Hence, there is no difference in both models regarding the time for loading an object. The real difference lies in the times for unloading an object from the program context. Cache systems rely on "guessing" — via cache replacement algorithms such as LRU, FIFO, *etc*— for unloading an object. On the other hand, programmers in our language can specify the exact unloading time for an object. Consequently, the execution model no longer has to rely on those ad-hoc replacement algorithms and memory can be managed in a more efficient way.

5 ACKNOWLEDGEMENTS

This paper was supported by DongA university research fund in 2001.

REFERENCES

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[Meyer] Bertrand Meyer. *Object-Oriented Software Construction. 2nd edition.* Prentice-Hall, 1997.

[Mil89a] Dale Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

[Mil89b] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79 – 108, 1989.

About the authors

Keehang Kwon is an associate professor at DongA university, Korea. He can be reached at keehang0@yahoo.co.kr.