LETTER

Towards Interactive Object-Oriented Programming

Keehang KWON[†a)], **Kyunghwan PARK**[†b)], *Nonmembers*, *and* **Mi-Young PARK**[†c)], *Member*

SUMMARY To represent interactive objects, we propose a choice-disjunctive declaration statement of the form $S \sqcup R$ where S, R are the (procedure or field) declaration statements within a class. This statement has the following semantics: request the user to choose one between S and R when an object of this class is created. This statement is useful for representing interactive objects that require interaction with the user.

key words: *interactions, object-oriented, computability logic*

1. Introduction

Interactive programming [4], [5] is an important modern trend in information technology. Despite the attention, object-oriented languages [6]–[8] have traditionally lacked mechanisms for representing interactive objects. For example, an object like a lottery ticket is in a superposition state of several possible values and require further interactions with the environment to determine their final value.

To represent interactive objects, we propose to adopt a choice-disjunctive operator in computability logic [1], [2]. To be specific, we allow, within a class definition, a choice-disjunctive declaration statement of the form $S \sqcup R$. This statement has the following semantics: request the user to choose one between S and R when an object is created. This statement is useful for representing interactive objects. For example, a lottery ticket, declared as $value = \$0 \sqcup value = \$1M$, indicates that it has two possible values, nothing or one million dollars, and its final value will be determined by the environment (or the user).

As a more complex example, consider a car rental system which is declared as

$$bmw = 320 \sqcup honda = civic \sqcup honda = accord.$$

This system is in a superposition state of several possible $fields$ and requires the user to determine its final field (the car maker) and its value (the model). Note that this system is very difficult to encode in Java because the field itself is only determined at run-time.

There are many objects which requires a form of bounded-choice interactions; the user is expected to choose one among many alternatives. Examples include most interactive objects such as airline ticketing systems and McDonalds. These objects may be encoded into traditional Java objects, but these encodings are typically complex, indirect and very awkward due to the dynamic nature of interactive objects.

The use of $\sqcup$ thus provides us a mechanism with which we can represent most interactive objects in an elegant way. This is, as far as we know, the first attempt to providing interactive features to objects in object-oriented languages. Hence, the major advantage of our proposal over Java is that it can optimally encode interactive objects which require the user to choose one among many alternatives.

The remainder of this paper is structured as follows. We describe the new language $Java^i$ in the next section. In Sect. 3, we present some examples. Section 4 concludes the paper.

2. The Language

The language is a subset of the core (untyped) Java with some extensions. It is described by G- and D-formulas given by the syntax rules below:

$$G ::= \ A \mid x = E \mid G; G \mid o = new\ D$$
$$D ::= \ A := G \mid x = E \mid \forall x\ D \mid D \wedge D \mid D \sqcup D$$

In the rules above, o is an object name, x is a field name, E is an expression, and A represents a procedure (or a method) of the form $p(t_1, \ldots, t_n)$. The notation $x = E$ in G denotes an assignment statement.

A D-formula is called a class definition. The notation $x = E$ in D denotes a field x with an initial value E. The notation $A := G$ in D denotes a procedure declaration where G is called a procedure body. The notation $D \wedge D$ denotes a conjunction of two D-formulas.

In the transition system to be considered, G-formulas will function as the main program (or procedure bodies), and a set of tuples $\langle o, D\rangle$ where o is an object name and D is a D-formula will constitute a program.

We will present an operational semantics for this language via a proof theory. The rules are formalized by means of what it means to execute the main task G from a program $\mathcal{P}$. These rules in fact depend on the top-level constructor in the expression, a property known as uniform provability [3]. Below the notation $\langle o, D\rangle; \mathcal{P}$ denotes $\{\langle o, D\rangle\} \cup \mathcal{P}$ but with the $\langle o, D\rangle$ tuple being distinguished (marked for backchaining). Note that execution alternates between two phases: the goal-reduction phase (one without a distinguished tuple) and the backchaining phase (one with a distinguished tuple). The notation $S\ sand\ R$ denotes the following: execute

Manuscript received February 20, 2013.
Manuscript revised October 2, 2014.
[†]The authors are with Computer Eng., DongA Univ., Korea.
a) E-mail: khkwon@dau.ac.kr
b) E-mail: khpark@dau.ac.kr (Corresponding author)
c) E-mail: openmp@dau.ac.kr
DOI: 10.1587/transinf.2013EDL8047

S and execute R sequentially. It is considered a success if both executions succeed. The notation $o.G$ represent an association of o with every field or procedure name appearing in G. For example, if G is $p(t_1, \ldots, t_n)$, then $o.G$ represents $o.p(t_1, \ldots, t_n)$.

Definition 1. Let o be an object name, let G be a main task and let $\mathcal{P}$ be a program. Then the notion of executing $\langle \mathcal{P}, o.G \rangle$ successfully and producing a new program $\mathcal{P}'$ – $ex(\mathcal{P}, o.G, \mathcal{P}')$ – is defined as follows:

(1) $ex(\langle o, (A := G) \rangle; \mathcal{P}, A, \mathcal{P}')$ if $ex(\mathcal{P}, o.G, \mathcal{P}')$. % matching procedure for A is found

(2) $ex(\langle o, \forall x D \rangle; \mathcal{P}, A, \mathcal{P}')$ if $ex(\langle o, [t/x]D \rangle; \mathcal{P}, A, \mathcal{P}')$. % argument passing

(3) $ex(\langle o, D_1 \wedge D_2 \rangle; \mathcal{P}, A, \mathcal{P}')$ if $ex(\langle o, D_1 \rangle; \mathcal{P}, A, \mathcal{P}')$. % looking for the procedure A in D_1.

(4) $ex(\langle o, D_1 \wedge D_2 \rangle; \mathcal{P}, A, \mathcal{P}')$ if $ex(\langle o, D_2 \rangle; \mathcal{P}, A, \mathcal{P}')$. % looking for the procedure A in D_2

(5) $ex(\mathcal{P}, o.A, \mathcal{P}')$ if $\langle o, D \rangle \in \mathcal{P}$ and $ex(\langle o, D \rangle; \mathcal{P}, A, \mathcal{P}')$. % a procedure call in object o

(6) $ex(\mathcal{P}, o.x = E, \mathcal{P}')$ where $\mathcal{P}'$ is obtained from $\mathcal{P}$ by first evaluating E to E' and updating the value of the field x to E' in the object o.

(7) $ex(\mathcal{P}, G_1; G_2, \mathcal{P}_2)$ if $ex(\mathcal{P}, G_1, \mathcal{P}_1)$ sand $ex(\mathcal{P}_1, G_2, \mathcal{P}_2)$.

(8) $ex(\mathcal{P}, o = new\ D, \{\langle o, D' \rangle\} \cup \mathcal{P})$ where D' is obtained from D by first removing choice-disjunctions and then by initializing its fields. % object creation

If $ex(\mathcal{P}, G, \mathcal{P}_1)$ has no derivation, then the machine returns the failure. In the above, the rules (1) to (4) deal with the backchaining phase, whereas the rules (5) to (8) deal with the goal reduction phase. Our operational semantics is a standard one appearing in most textbooks. Only the rule (8) is a novel feature.

3. Examples

Imagine Temple University charges \$5,000 as its tuition for nonemployees and \$3,000 for employees. An example of this class is provided by the following program:

$class\ TempleU$
$tuition = 0 \wedge$
$(employee = true \sqcup employee = false) \wedge$
$(comp\_tuition() := if\ employee\ then\ tuition = \$3000$
$\qquad else\ tuition = \$5000)$
$void\ main()$
$TempleU\ p = new\ TempleU;$
$comp\_tuition();$
$print(p.tuition)$

In the above, creating a TempleU object via the *new* construct basically proceeds as follows: the machine asks the user "are you an employee?". If the user answers yes by choosing the left disjunct, *employee* will be initialized to *true* and the machine will eventually print \$3000 for its tuition. If the user answers no by choosing the right disjunct, *employee* will be initialized to $false$ and the machine will eventually print \$5000 for its tuition. Our language thus makes it possible to customize the amount for tuition via interaction with the user.

4. Conclusion

In this paper, we extend the core Java with the addition of disjunctive statements within a class definition. This extension allows statements of the form $S \sqcup R$ where S, R are statements. These statements are particularly useful for representing interactive objects.

Acknowledgements

This work was supported by Dong-A University Research Fund.

References

[1] G. Japaridze, "Introduction to computability logic," Annals of Pure and Applied Logic, vol.123, pp.1–99, 2003.
[2] G. Japaridze, "Sequential operators in computability logic," Information and Computation, vol.206, no.12, pp.1443–1475, 2008.
[3] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, "Uniform proofs as a foundation for logic programming," Annals of Pure and Applied Logic, vol.51, pp.125–157, 1991.
[4] L.A. Stein, "Interactive programming: Revolutionizing introductory computer science," ACM Comput. Surv. 28, 4es, Article no.103, Dec. 1996.
[5] R. Perera, "First-order interactive programming," 12th International Conference on Practical Aspects of Declarative Languages (PADL'10), pp.186–200, Jan. 2010.
[6] A.C. Kak, Programming with Objects: A Comparative Presentation of Object-Oriented Programming with C++ and Java, John Wiley & Sons, New York, NY, USA, 2003.
[7] J. Albahari and B. Albanhari, C# 5.0 Pocket Reference, O'Reilly Media, 224 pages, May 2012.
[8] J. Bloch, Effective Java, second Ed., Addison-Wesley, 2008.