π-calculus Semantics of Object-Oriented Programming Languages

π-calculus Semantics of Object-Oriented Programming Languages

by

David Walker

LFCS Report Series ECS-LFCS-90-122

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

October 1990

π-calculus Semantics of Object-Oriented Programming Languages

David Walker
University of Technology, Sydney

September 1990

Introduction

The π-calculus provides a rigorous foundation for the study of concurrent systems with evolving communication structure. [7] contains an introduction to the calculus through a sequence of examples, while in [8] its elementary theory is developed in detail. In addition to the examples in [7], evidence of the expressive power of the calculus is provided by [6], in which encodings of λ-calculus in the π-calculus are studied; by [9], where a communications protocol used in a mobile telephone network is modelled; and by [11] which contains a translation of CCS with value-passing into the π-calculus. In the present paper we develop this theme further, providing semantics for a pair of object-oriented programming languages by translating them into the π-calculus. The translations build on earlier work on the π-calculus and the translation of a parallel imperative language into CCS from [5]. They illustrate how the calculus may be used to provide succinct, natural and illuminating representations of systems described in an object-oriented style.

The first language considered is a mild enrichment of the language P of [4]; the second is closely related to the parallel object-oriented language POOL [1]. The languages differ essentially from one another only in the mechanisms they provide for interaction between objects: in the first language, interaction occurs via synchronous communication of data; in the second, a rendezvous mechanism is employed. The encodings of these mechanisms in the π-calculus are very close to natural operational intuitions.

It is interesting to compare the semantics offered here with alternative semantics of related languages, for example the denotational semantics for POOL using metric spaces in [3]. One point worthy of note is that by providing the semantics by translating into the π-calculus, we avoid the need to wield any heavy mathematical machinery in establishing the well-definedness of the semantics. Further the translation provides, amongst other things, a clear view of the central concept: the object. It is interesting also to note how a few natural ideas, such as replication and the use of private communication links, may be used to model data values, methods and classes uniformly by viewing them as generators of instances of appropriate kinds of entity.

Several variations of the translations have been examined; we believe it is not easy to find translations notably simpler than those presented. This paper contains only the translations: no detailed investigation of their properties is included. This investigation, together with study of the relationships of the semantics to those given by other means, for instance the semantics for POOL in [2] and [10], remain topics for future research.

This paper assumes familiarity with the π-calculus as presented in the papers cited above. In particular we adopt the notational conventions of [7].

1 The languages

A program of either of the languages we consider describes the computational behaviour of a system of entities called *objects*. In the first language $\mathcal{L}_1$, which is a mild enrichment of the language P of [4], an object consists of a sequential statement together with a family of variables local to the object. Variables may contain data values such as integers or references to objects. The second language $\mathcal{L}_2$ is closely related to the parallel object-oriented language POOL [1]. In addition to the features of an $\mathcal{L}_1$-object, an $\mathcal{L}_2$-object possesses a number of procedures or *methods*.

The languages differ essentially from one another only in the mechanisms they provide for interaction between objects. In $\mathcal{L}_1$, objects interact via a synchronous communication mechanism in which a data value is sent from one object to another and assigned to a variable of the receiving object. In order for such an interaction to be possible, one of the sending object's variables must contain a reference to the receiving object. There are two modes of reception of a data value. In the first, one of the receiving object's variables must contain a reference to the sending object, while in the second the receiving object may accept a value (of the appropriate type) from any object. Interaction between objects in $\mathcal{L}_2$ occurs via a rendezvous mechanism in which a method is invoked in one object on receipt of a message from a second object. The sending object's activity is suspended until the method invocation is completed and a value returned to it by the receiving object.

Objects may be created during a computation. Moreover the communication structure among objects may evolve through the communication of references to objects. Each object is an instance of some *class*, and a program consists of a sequence of class declarations and a (conventional) indication of which of these classes furnishes the *root object* which alone exists at the initiation of a computation. We now describe the languages in detail, beginning with $\mathcal{L}_1$.

The language $\mathcal{L}_1$

We begin with a set of program variables ranged over by X, Y, Z. The class of $\mathcal{L}_1$-expressions ranged over by E is given as follows:

$$
\begin{array}{rcll}
E & ::= & X & \\
 & \mid & k & (k = 0, 1, 2, \ldots) \\
 & \mid & b & (b = \text{true, false}) \\
 & \mid & \mathbf{nil} & \\
 & \mid & \mathbf{self} & \\
 & \mid & E_1 + E_2 & \\
 & \mid & E_1 = E_2 & \\
 & \mid & \neg E & \\
 & \mid & E_1 \wedge E_2 &
\end{array}
$$

Each well-formed expression is of one of the types *nat*, *bool* and *ref*; the type of a variable is given via a declaration. **nil** and **self** are of type *ref*; **nil** signifies a reference to no object while, loosely speaking, **self** signifies the object in which it occurs. This simple language of expressions is adequate for illustrating the principal points of interest in the encoding. Richer languages may be accommodated by extending the techniques described below; see also [7] and [11].

We assume further a set of class names ranged over by C. The class of $\mathcal{L}_1$-statements ranged over by S is given as follows:

$$
\begin{array}{rcll}
S & ::= & \mathbf{skip} & \\
 & \mid & X := E & \\
 & \mid & X := \mathbf{new}_C & (C \text{ a class name}) \\
 & \mid & X!E & \\
 & \mid & X?Y & \\
 & \mid & ?Y & \\
 & \mid & S_1; S_2 & \\
 & \mid & \mathbf{if}\ E\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 & \\
 & \mid & \mathbf{while}\ E\ \mathbf{do}\ S &
\end{array}
$$

Statements of the forms $X := \mathbf{new}_C$, $X!E$ and $X?Y$ are well-formed only if X is of type *ref*. The effect of $X := \mathbf{new}_C$ is to create a new object of class C and to assign to X a reference to it. In $X!E$ the value of the expression E is sent to the object to which X contains a reference (if it exists), while in $X?Y$ a value is received from the object to which X contains a reference (if it exists) and assigned to Y. In $?Y$ a value is received from some object and assigned to Y. In any communication arising from $X!E$ and $Y?Z$ or from $X!E$ and $?Z$, the types of E and Z must agree. The remaining constructs have their usual meanings. $X := E$ is well-formed only if X and E share a common type; **if** E **then** S_1 **else** S_2 and **while** E **do** S are well-formed only if E is of type *bool*.

The class of $\mathcal{L}_1$-declarations is given as follows with t ranging over types. First there are sequences of variable declarations:

$$\mathit{Vdec} \quad ::= \quad \mathbf{var}\ X_1 : t_1, \ldots, X_n : t_n$$

with the X_i distinct from one another. Next there are class declarations:

$$\mathit{Cdec} \quad ::= \quad \mathbf{class}\ C\ \mathbf{is}\ \mathit{Vdec}\ \mathbf{in}\ S$$

with every variable occurring in S being declared in *Vdec*. And finally there are program declarations:

$$Pdec \ ::= \ \textbf{program}\ P\ \textbf{is}\ Cdec_1, \ldots, Cdec_n$$

where if $Cdec_i \equiv \textbf{class}\ C_i\ \textbf{is}\ Vdec_i\ \textbf{in}\ S_i$, then: the C_i are distinct from one another; and if $X := \textbf{new}_{C'}$ occurs in one of the S_i then C' is one of the C_j. By convention, C_1 is the class of the single root object which alone exists at the initiation of a computation.

On creation, an object augments the computation by executing the statement appearing in the declaration of the class of which it is an instance with its own family of local variables determined by the variable declaration part of the class declaration.

The language $\mathcal{L}_2$

$\mathcal{L}_2$ shares many features with $\mathcal{L}_1$. To describe its expressions we assume a set of method names ranged over by M. The class of $\mathcal{L}_2$-expressions is obtained by augmenting the definition of the class of $\mathcal{L}_1$-expressions with one extra clause:

$$\begin{array}{rcl} E & ::= & \ldots \\ & | & E_1!M(E_2) \end{array}$$

An expression of the form $E_1!M(E_2)$ is well-formed only if E_1 is of type *ref*. If the value of E_1 is a reference to an object, the value of the expression is the value returned by that object after an invocation of the method M with parameter E_2. Such an invocation may occur if the object referred to executes an **answer** statement as described below.

The class of $\mathcal{L}_2$-statements is obtained by deleting from the definition of the class of $\mathcal{L}_1$-statements the clauses for $X!E$, $X?Y$ and $?Y$, and adding instead one new clause:

$$\begin{array}{rcl} S & ::= & \ldots \\ & | & \textbf{answer}(M_1, \ldots, M_k) \end{array}$$

A statement $\textbf{answer}(M_1, \ldots, M_k)$ requires that for some i with $1 \leq i \leq k$, the statement comprising the body of the declaration of M_i (see below) be executed with a parameter supplied by an object seeking to evaluate an expression $E_1!M_i(E_2)$ with E_1 a reference to the object containing the **answer** statement. On termination of the method invocation a value is returned to the sending object, this being the value of the expression $E_1!M_i(E_2)$. Note that under this interpretation an $\textbf{answer}(M_1, \ldots, M_k)$ statement requires that an appropriate method be invoked. An alternative in which if no invocation requests are outstanding the **answer** statement may successfully terminate is also possible. With a little more work this too may be encoded in the π-calculus.

The class of $\mathcal{L}_2$-declarations contains in addition to sequences of variable declarations, sequences of method declarations of the form:

$$Mdec \ ::= \ \textbf{method}\ M_1(X_1, Y_1)\ \textbf{is}\ S_1, \ldots, M_n(X_n, Y_n)\ \textbf{is}\ S_n$$

with the M_i distinct from one another. In a method declaration $\textbf{method}\ M(X, Y)\ \textbf{is}\ S$, X is the formal parameter supplied by an invoking expression $E_1!M(E_2)$, and Y a variable in which will be stored the value to be returned as the value of that expression on completion of the execution of the body S. We consider only one-parameter methods, the extension of the translation to the many-parameter case being quite straightforward.

The clause for $\mathcal{L}_2$-class declarations is:

$$\mathit{Cdec} \quad ::= \quad \textbf{class}\ C\ \textbf{is}\ \mathit{Vdec},\ \mathit{Mdec}\ \textbf{in}\ S$$

with every variable and every method name occurring in S being declared in Vdec and Mdec.

Finally, the clause for $\mathcal{L}_2$-program declarations is:

$$\mathit{Pdec} \quad ::= \quad \textbf{program}\ P\ \textbf{is}\ \mathit{Cdec}_1, \ldots, \mathit{Cdec}_n$$

with caveats and interpretation similar those in the case of $\mathcal{L}_1$.

2 The translations

Each entity of each syntactic category is represented as an agent[1], the representations of complex entities being constructed from those of their constituents. There are delicate relationships between certain parts of the translations, and for this reason some details may not be entirely clear when read in isolation. We begin with the translation of $\mathcal{L}_1$.

The translation of $\mathcal{L}_1$

To give a feel for the overall structure of the translation we begin with declarations. First we consider variables of non-reference types. We assume that for each variable X there are constant names r_X and w_X. As discussed in [7], the introduction of constant names may be avoided. However rather than introducing a further level of encoding we admit them here.

With t being either *nat* or *bool* we define:

$$[\![\textbf{var}\ X : t]\!] \overset{\text{def}}{=} \mathit{Loc}_X$$

where

$$\begin{aligned} \mathit{Loc}_X &\overset{\text{def}}{=} w_X(y).\, \mathit{Reg}_X(y) \\ \mathit{Reg}_X(y) &\overset{\text{def}}{=} \overline{r_X}y.\, \mathit{Reg}_X(y) + w_X(z).\, \mathit{Reg}_X(z) \end{aligned}$$

Loc_X represents a memory location at which the name of a link to an agent representing a value may be stored. Such a name may be written via the link w_X, while the name currently stored may be read via the link r_X. Assignment to X will be represented by the storing of the name of a link to the agent encoding the new data value. Note that on its declaration no link name is stored; an attempt to read from the variable will fail.

[1]more accurately, as a mapping from names to agents

The translation of the declaration of a variable of type *ref* is slightly different. However it is convenient to use the same agent constants relying on context to resolve any ambiguity. We use two further constant names: NIL and REF. We define:

$$[\![\mathbf{var}\ X : \mathit{ref}]\!] \overset{\mathrm{def}}{=} \mathit{Loc}_X$$

where

$$\begin{aligned} \mathit{Loc}_X &\overset{\mathrm{def}}{=} \overline{r_X}\,\mathtt{NIL}.\,\mathit{Loc}_X + w_X : [\mathtt{NIL} \Rightarrow \mathit{Loc}_X,\ \mathtt{REF} \Rightarrow w_X(y).\,\mathit{Reg}_X(y)] \\ \mathit{Reg}_X(y) &\overset{\mathrm{def}}{=} \overline{r_X}\,\mathtt{REF}.\,\overline{r_X}y.\,\mathit{Reg}_X(y) + w_X : [\mathtt{NIL} \Rightarrow \mathit{Loc}_X,\ \mathtt{REF} \Rightarrow w_X(z).\,\mathit{Reg}_X(z)] \end{aligned}$$

REF and NIL signify, respectively, that X does or does not contain a reference to an object; note the use of the match construct in the definitions. Otherwise the behaviour is similar to the case of X of non-reference type. Note that on its declaration X is deemed to contain no reference to an object.

A sequence of variable declarations is represented as the composition of the agents representing the individual declarations:

$$[\![\mathbf{var}\ X_1 : t_1, \ldots, X_n : t_n]\!] \overset{\mathrm{def}}{=} [\![\mathbf{var}\ X_1 : t_1]\!] \mid \cdots \mid [\![\mathbf{var}\ X_n : t_n]\!]$$

Next we consider class declarations. We assume for each class name C a constant name c of the π-calculus. The translation is:

$$[\![\mathbf{class}\ C\ \mathbf{is}\ \mathit{Vdec}\ \mathbf{in}\ S]\!](c) \overset{\mathrm{def}}{=} \overline{c}(w) * [\![\mathit{Vdec}\ \mathbf{in}\ S]\!](w)$$

where

$$[\![\mathit{Vdec}\ \mathbf{in}\ S]\!](w) \overset{\mathrm{def}}{=} (N)([\![\mathit{Vdec}]\!] \mid [\![S]\!](w))$$

with $N = \{r_X,\ w_X \mid X \text{ occurs in } \mathit{Vdec}\}$. Recall from [7] that $\alpha * P$ is a *replicator.*

As explained below, each statement S is represented as an agent $[\![S]\!](w)$ with w playing the rôle of a reference to the object of which S is a part. The agent representing the class declaration may provide at c an indefinite number of private instances of the agent $[\![\mathit{Vdec}\ \mathbf{in}\ S]\!]$ which represents an object with statement agent $[\![S]\!]$ composed with agents representing the local variables. Note the rôle of the restriction operator in localizing the scope of the variables. Note also the means by which the agent representing the class declaration produces a new instance of the class accessible only to the recipient of the private name w. As explained below, the recipient will be an agent representing the expression $\mathbf{new}_C$.

Finally if $\mathit{Pdec} \equiv \mathbf{program}\ P\ \mathbf{is}\ \mathit{Cdec}_1, \ldots, \mathit{Cdec}_n$ is a program declaration, its translation is as follows:

$$[\![\mathit{Pdec}]\!](w) \overset{\mathrm{def}}{=} (c_1, \ldots, c_m)([\![\mathit{Cdec}_1]\!](c_1) \mid \cdots \mid [\![\mathit{Cdec}_n]\!](c_n) \mid c_1(w).\,\mathbf{0})$$

Note how the initiation of the computation with the single root object of class C_1 is represented using the trigger $c_1(w).\,\mathbf{0}$.

Next we consider the translation of expressions of non-reference type. Each such expression E is represented as an agent $[\![E]\!](v)$ with v the name of a link through which the agent may communicate its value. Such agents have a transient existence: they may yield their value only once. However as explained below, more permanent forms are also used in the translation. We assume four more constant names: ZERO, ONE, TRUE and FALSE. We have:

$$\begin{aligned}
[\![k]\!](v) &\overset{\text{def}}{=} (\overline{v}\,\mathtt{ONE}.\,)^{k}\,\overline{v}\,\mathtt{ZERO} \\
[\![\mathit{true}]\!](v) &\overset{\text{def}}{=} \overline{v}\,\mathtt{TRUE} \\
[\![\mathit{false}]\!](v) &\overset{\text{def}}{=} \overline{v}\,\mathtt{FALSE} \\
[\![E_1 + E_2]\!](v) &\overset{\text{def}}{=} (v_1, v_2)([\![E_1]\!](v_1) \mid [\![E_2]\!](v_2) \mid [\![+]\!](v_1, v_2, v)) \\
[\![E_1 = E_2]\!](v) &\overset{\text{def}}{=} (v_1, v_2)([\![E_1]\!](v_1) \mid [\![E_2]\!](v_2) \mid [\![=]\!](v_1, v_2, v)) \\
[\![\neg E]\!](v) &\overset{\text{def}}{=} (v_1)([\![E_1]\!](v_1) \mid [\![\neg]\!](v_1, v)) \\
[\![E_1 \wedge E_2]\!](v) &\overset{\text{def}}{=} (v_1, v_2)([\![E_1]\!](v_1) \mid [\![E_2]\!](v_2) \mid [\![\wedge]\!](v_1, v_2, v))
\end{aligned}$$

where

$$\begin{aligned}
[\![+]\!](v_1, v_2, v) &\overset{\text{def}}{=} v_1 : [\mathtt{ZERO} \Rightarrow [\![+]\!]'(v_2, v),\ \mathtt{ONE} \Rightarrow \overline{v}\,\mathtt{ONE}.\,[\![+]\!](v_1, v_2, v)] \\
[\![+]\!]'(v_2, v) &\overset{\text{def}}{=} v_2 : [\mathtt{ZERO} \Rightarrow \overline{v}\,\mathtt{ZERO},\ \mathtt{ONE} \Rightarrow \overline{v}\,\mathtt{ONE}.\,[\![+]\!]'(v_2, v)] \\
[\![=]\!](v_1, v_2, v) &\overset{\text{def}}{=} v_1 : [\mathtt{ZERO} \Rightarrow v_2 : [\mathtt{ZERO} \Rightarrow \overline{v}\,\mathtt{TRUE},\ \mathtt{ONE} \Rightarrow \overline{v}\,\mathtt{FALSE}], \\
&\qquad\quad \mathtt{ONE} \Rightarrow v_2 : [\mathtt{ZERO} \Rightarrow \overline{v}\,\mathtt{FALSE},\ \mathtt{ONE} \Rightarrow [\![=]\!](v_1, v_2, v)]] \\
[\![\neg]\!](v_1, v) &\overset{\text{def}}{=} v_1 : [\mathtt{FALSE} \Rightarrow \overline{v}\,\mathtt{TRUE},\ \mathtt{TRUE} \Rightarrow \overline{v}\,\mathtt{FALSE}] \\
[\![\wedge]\!](v_1, v_2, v) &\overset{\text{def}}{=} v_1 : [\mathtt{FALSE} \Rightarrow \overline{v}\,\mathtt{FALSE},\ \mathtt{TRUE} \Rightarrow v_2(y).\,\overline{v}y]
\end{aligned}$$

Note that in these definitions we abbreviate $\alpha.\,\mathbf{0}$ to α.

For X of non-reference type we have:

$$[\![X]\!](v) \overset{\text{def}}{=} r_X(y).\,y(u).\,\mathit{Copy}(u, v)$$

where

$$\mathit{Copy}(u, v) \overset{\text{def}}{=} u(z).\,\overline{v}z.\,\mathit{Copy}(u, v)$$

Some explanation of this representation is required. However, it is much more readily appreciated in conjunction with the translation of the assignment statement $X := E$, so we proceed to this first. Three preparatory steps are required.

The first involves the introduction of a further constant name *done* and the following definition:

$$\mathit{Done} \overset{\text{def}}{=} \overline{\mathit{done}}\,\mathit{done}.\,\mathbf{0}$$

The rôle of *Done* is to provide a signal of successful termination of a statement. This will be elaborated below.

The second step is the introduction of a derived operator *before* as follows:

$$P \textit{ before } Q \overset{\text{def}}{=} (\mathit{done})(P \mid \mathit{done}(z).\,Q) \quad \text{where } z \notin \mathit{fn}(Q)$$

Finally the third step involves the introduction of a family of agents representing the evaluation of expressions (of all types). We set:

$$\begin{array}{rcl} \mathit{Eval}(v_1, v) & \stackrel{\text{def}}{=} & v_1 : [\ \begin{array}{l} \mathtt{ZERO} \Rightarrow \mathit{Nateval}_0(v), \\ \mathtt{ONE} \Rightarrow \mathit{Nateval}_1(v_1, v), \\ \mathtt{FALSE} \Rightarrow \mathit{Booleval}_F(v), \\ \mathtt{TRUE} \Rightarrow \mathit{Booleval}_T(v), \\ \mathtt{NIL} \Rightarrow \mathit{Refeval}_N(v), \\ \mathtt{REF} \Rightarrow \mathit{Refeval}_R(v_1, v)] \end{array} \end{array}$$

where

$$\begin{array}{rcll} \mathit{Nateval}_0(v) & \stackrel{\text{def}}{=} & \mathit{Done} \mid \overline{v}(w) * [\![0]\!](w) & \\ \mathit{Nateval}_k(v_1, v) & \stackrel{\text{def}}{=} & v_1 : [\mathtt{ZERO} \Rightarrow \mathit{Done} \mid \overline{v}(w) * [\![k]\!](w), & \\ & & \qquad \mathtt{ONE} \Rightarrow \mathit{Nateval}_{k+1}(v_1, v)] & \text{if } k \geq 1 \\ \mathit{Booleval}_F(v) & \stackrel{\text{def}}{=} & \mathit{Done} \mid \overline{v}(w) * [\![\mathit{false}]\!](w) & \\ \mathit{Booleval}_T(v) & \stackrel{\text{def}}{=} & \mathit{Done} \mid \overline{v}(w) * [\![\mathit{true}]\!](w) & \\ \mathit{Refeval}_N(v) & \stackrel{\text{def}}{=} & \mathit{Done} \mid \overline{v}(w) * \mathit{Nil}(w) & \\ \mathit{Refeval}_R(v_1, v) & \stackrel{\text{def}}{=} & v_1(u).\,(\mathit{Done} \mid \overline{v}(w) * \mathit{Ref}(w, u)) & \end{array}$$

where

$$\begin{array}{rcl} \mathit{Nil}(w) & \stackrel{\text{def}}{=} & \overline{w}\,\mathtt{NIL} \\ \mathit{Ref}(w, u) & \stackrel{\text{def}}{=} & \overline{w}\,\mathtt{REF}.\,\overline{w}u \end{array}$$

Note in passing that the encoding of data values may be extended easily to arbitrary sets of values built recursively from a finite set of constructors.

Then for X and E of common non-reference type we have:

$$[\![X := E]\!](w) \stackrel{\text{def}}{=} (v)((v_1)([\![E]\!](v_1) \mid \mathit{Eval}(v_1, v))\ \mathit{before}\ \overline{w_X}v.\,\mathit{Done})$$

Thus the assignment is represented by the evaluation of E followed by the passing to Reg_X of a private link to the replicator representing the value of E and a signal (to the agent representing the subsequent command) of successful termination. Note from the definition of *Eval* that an indefinite number of private instances of the agent representing the value of E may be obtained via the link stored in Reg_X.

The definition of $[\![X]\!]$ may now be clearer. The evaluation of an expression X (of non-reference type) is represented by the reception from Reg_X of the name of a private link to the replicator representing the current value of X, followed by the reception from the replicator of the name of a private link to an agent capable of yielding the value, followed in turn by the transmission of the value piecemeal. To gain some feel for the representation the reader may care to examine the encoding of a fragment such as:

$$\textbf{var}\ X : \mathit{nat},\ Y : \mathit{nat}\ \textbf{in}\ X := 3;\ Y := X + X$$

The translation of the sending statement is similar. We have:

$$\begin{array}{rcl} [\![X!E]\!](w) & \stackrel{\text{def}}{=} & r_X(u).\,[u = \mathtt{REF}]\,r_X(x). \\ & & (v)((v_1)([\![E]\!](v_1) \mid \mathit{Eval}(v_1, v))\ \mathit{before}\ \overline{x}v.\,\mathit{Done}) \end{array}$$

First the name x of the link to the agent representing the receiving object is read from Reg_X. Then E is evaluated and the private link to the replicated version of its value passed along x.

The receiving statements are represented as follows:

$$[\![X?Y]\!](w) \overset{\mathrm{def}}{=} r_X(u).\,[u = \mathtt{REF}]r_X(x).\,x(y).\,\overline{w_Y}y.\,\mathit{Done}$$

in which the name x of the link to the sending object is read from Reg_X, the name y of the private link to the replicated version of the incoming data value is received along x, and this name is sent to Reg_Y. Secondly:

$$[\![?Y]\!](w) \overset{\mathrm{def}}{=} w(y).\,\overline{w_Y}y.\,\mathit{Done}$$

Here one point of the parameter w in the representation of statements is seen. The reception of a value from an unspecified object is represented by the reception of a name along the link named w. The other rôle of this parameter is in the translation of the expression **self** as described below.

The translation of expressions of type *ref* is as follows:

$$\begin{aligned}
[\![\mathbf{nil}]\!](w,v) &\overset{\mathrm{def}}{=} \overline{v}\,\mathtt{NIL}\\
[\![\mathbf{self}]\!](w,v) &\overset{\mathrm{def}}{=} \overline{v}\,\mathtt{REF}.\,\overline{v}w\\
[\![X]\!](w,v) &\overset{\mathrm{def}}{=} \mathit{Copy}(r_X, v)
\end{aligned}$$

Note the presence of two parameters in these definitions. The first of these will be the parameter in the translation $[\![S]\!](w)$ of the statement in which the expression in question occurs. Also, compare the representation $[\![X]\!]$ with the corresponding definition for X of non-reference type. In this case no passing of private links to copies of agents representing data values is required.

The translations of the assignment, sending and receiving statements for reference types are as follows:

$$\begin{aligned}
[\![X := E]\!](w) \overset{\mathrm{def}}{=}\ & (v)([\![E]\!](w,v) \mid v : [\ \mathtt{NIL} \Rightarrow \overline{w_X}\,\mathtt{NIL}.\,\mathit{Done},\\
& \qquad \mathtt{REF} \Rightarrow \overline{w_X}\,\mathtt{REF}.\,v(u).\,\overline{w_X}u.\,\mathit{Done}])
\end{aligned}$$

$$\begin{aligned}
[\![X!E]\!](w) \overset{\mathrm{def}}{=}\ & (v)([\![E]\!](w,v) \mid r_X(u).\,[u = \mathtt{REF}]\,r_X(x).\,(y)\overline{x}y.\\
& v : [\mathtt{NIL} \Rightarrow \overline{y}\,\mathtt{NIL}.\,\mathit{Done},\ \mathtt{REF} \Rightarrow \overline{y}\,\mathtt{REF}.\,v(u).\,\overline{y}u.\,\mathit{Done}])
\end{aligned}$$

$$\begin{aligned}
[\![X?Y]\!](w) \overset{\mathrm{def}}{=}\ & r_X(u).\,[u = \mathtt{REF}]\,r_X(x).\,x(y).\\
& y : [\mathtt{NIL} \Rightarrow \overline{w_Y}\,\mathtt{NIL}.\,\mathit{Done},\ \mathtt{REF} \Rightarrow \overline{w_Y}\,\mathtt{REF}.\,y(u).\,\overline{w_Y}u.\,\mathit{Done}]
\end{aligned}$$

$$\begin{aligned}
[\![?Y]\!](w) \overset{\mathrm{def}}{=}\ & w(y).\,y : [\ \mathtt{NIL} \Rightarrow \overline{w_Y}\,\mathtt{NIL}.\,\mathit{Done},\\
& \qquad \mathtt{REF} \Rightarrow \overline{w_Y}\,\mathtt{REF}.\,y(u).\,\overline{w_Y}u.\,\mathit{Done}]
\end{aligned}$$

Compare these clauses with the corresponding representations in the case of non-reference types. A value of type *ref* is communicated via a *molecular action* (see [7]). A more uniform treatment of the encoding of expressions is possible. We adopt the present approach in order to illustrate the possibilities and also because the alternative does not appear to be simpler.

The final assignment involves the creation of a new object:

$$[\![X := \mathbf{new}_C]\!](w) \overset{\text{def}}{=} c(z).\, \overline{w_X}\,\mathtt{REF}.\, \overline{w_X}z.\, \mathit{Done}$$

Referring back to the translation of class declarations, this assignment is represented by the reception from the agent representing the declaration of C of a private link z to an agent representing a new object of class C followed by the sending of this link to Reg_X.

It remains only to deal with the standard statement constructs. We have:

$$\begin{aligned}
[\![\mathbf{skip}]\!](w) &\overset{\text{def}}{=} \mathit{Done} \\
[\![S_1; S_2]\!](w) &\overset{\text{def}}{=} [\![S_1]\!](w)\ \mathit{before}\ [\![S_2]\!](w) \\
[\![\mathbf{if}\ E\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2]\!](w) &\overset{\text{def}}{=} (v)([\![E]\!](v) \mid v : [\mathtt{TRUE} \Rightarrow [\![S_1]\!](w),\ \mathtt{FALSE} \Rightarrow [\![S_2]\!](w)]) \\
[\![\mathbf{while}\ E\ \mathbf{do}\ S]\!](w) &\overset{\text{def}}{=} W(w)
\end{aligned}$$

where

$$W(w) \overset{\text{def}}{=} (v)([\![E]\!](v) \mid v : [\mathtt{TRUE} \Rightarrow ([\![S]\!](w)\ \mathit{before}\ W(w)),\ \mathtt{FALSE} \Rightarrow \mathit{Done}])$$

These clauses are fairly self-explanatory (see [5]). This completes the translation of $\mathcal{L}_1$.

The translation of $\mathcal{L}_2$

The $\mathcal{L}_2$-translations of those constructs common to the two languages are identical to their $\mathcal{L}_1$-translations. The translations of the other constructs, method declarations, **answer** statements, and method invocation expressions, are intimately related to one another.

Suppose **method** $M(X,Y)$ **is** S is a single method declaration taking a parameter of non-reference type and returning a value of non-reference type. Then the translations are as follows where for each method name M we assume a constant name m of the π-calculus:

$$[\![\mathbf{method}\ M(X,Y)\ \mathbf{is}\ S]\!](m) \overset{\text{def}}{=} \overline{m}(z) * M(z)$$

where

$$M(z) \overset{\text{def}}{=} (N)(Loc_X \mid Loc_Y \mid z(w).\, z(x).\, \overline{w_X}x.\, ([\![S]\!](w)\ \mathit{before}\ r_Y(v').\, \overline{z}v'))$$

with $N = \{r_X, w_X, r_Y, w_Y\}$;

$$[\![\mathbf{answer}(M_1, \ldots, M_k)]\!](w) \overset{\text{def}}{=} w(u).\, u : [m_i \Rightarrow u(v).\, m_i(z).\, \overline{z}w.\, \overline{z}v.\, z(v').\, \overline{u}v'.\, \mathit{Done}]_{i=1}^{k}$$

and

$$
\begin{aligned}
[\![E_1!M(E_2)]\!](v) \stackrel{\text{def}}{=} (v_3)\ & ((v_1)([\![E_1]\!](v_1) \mid v_1(u)\,[u = \text{REF}]\,v_1(w).\,\mathit{Done}) \\
& \quad \mathit{before}\ ((v_2)([\![E_2]\!](v_2) \mid \mathit{Eval}(v_2, v_3)) \\
& \qquad \mathit{before}\ (u)\overline{w}u.\,\overline{u}m.\,\overline{u}v_3.\,u(v').\,v'(v'').\,\mathit{Copy}(v'', v)))
\end{aligned}
$$

The agent representing $E_1!M(E_2)$ first obtains the name of the link w to the agent representing the object to which E_1 refers (if it exists), and then the parameter E_2 is evaluated. Its activity is then suspended until the agent representing the object referred to is able to receive a communication along this link, indicating the execution of an **answer** statement. Then a private link u is communicated from the expression agent to the **answer** statement agent, and along this link there follows the name m representing the method M. If the method requested is among those offered, modelled using the match construct, the private link v_3 to the replicated version of the value of E_2 is sent along u, and then the **answer** agent requests from the agent representing the appropriate method declaration a private copy of the agent representing the method body (with private local variable agents). To this agent it communicates the parameter w indicating in which object it occurs – this is necessary as the body may contain **answer** statements or occurrences of the expression **self** – followed by the private link to the replicated value of the parameter. The method body agent stores this link in its local Reg_X, the agent $[\![S]\!]$ becomes active, and on completion the name stored in the local Reg_Y is returned to the **answer** agent. This agent in turn returns this name to the $E_1!M(E_2)$ expression agent, along the private link u previously established between them, and then indicates successful termination. Finally a link to a private copy of the value is obtained and the value is copied piecemeal via the *Copy* agent.

It is straightforward to modify the translation to handle the case when either the parameter or the value of the invoking expression is of type *ref*. Similarly, the representation of many-parameter methods is not difficult. As mentioned earlier, the interpretation of the **answer** statement adopted here is such that it may successfully terminate only after a method has been invoked. A representation of an alternative in which this is not the case is possible but is not given here.

If several objects are suspended awaiting an opportunity to invoke a method in a given object then according to the translation given here, when an **answer** statement is executed, only one will have this opportunity. Moreover if the method requested by the selected object is not among those offered in the **answer** statement, the agent representing the **answer** statement will be unable to proceed. An alternative in which incoming messages are stored in queues, with method invocations occurring in an order determined in some way by the queues' contents, may also be represented with a little more work.

It remains only to complete the translation of $\mathcal{L}_2$-declarations. The translation of a sequence $\mathit{Mdec}_1, \ldots, \mathit{Mdec}_n$ of method declarations with M_i the method name appearing in Mdec_i is:

$$[\![\mathit{Mdec}_1, \ldots, \mathit{Mdec}_n]\!](m_1, \ldots, m_n) \stackrel{\text{def}}{=} [\![\mathit{Mdec}_1]\!](m_1) \mid \cdots \mid [\![\mathit{Mdec}_n]\!](m_n)$$

The translation of an $\mathcal{L}_2$-class declaration is:

$$[\![\textbf{class}\ C\ \textbf{is}\ \mathit{Vdec},\ \mathit{Mdec}\ \textbf{in}\ S]\!](c) \stackrel{\text{def}}{=} \overline{c}(w) * [\![\mathit{Vdec},\ \mathit{Mdec}\ \textbf{in}\ S]\!](w)$$

where

$$[\![Vdec,\ Mdec\ \mathbf{in}\ S]\!](w) \overset{\text{def}}{=} (N)([\![Vdec]\!] \mid [\![Mdec]\!] \mid [\![S]\!](w))$$

with

$$N = \{r_X, w_X \mid X \text{ occurs in } Vdec\} \cup \{m \mid M \text{ occurs in } Mdec\}$$

Finally the translation of a program declaration is as for $\mathcal{L}_1$ with the above definitions. This completes the translation of $\mathcal{L}_2$.

References

[1] P. America, *Issues in the design of a parallel object-oriented language*, Formal Aspects of Computing, vol.1 no.4 pp.366–411 (1989).

[2] P. America, J. de Bakker, J. Kok and J. Rutten, *Operational semantics of a parallel object-oriented language*, in 13th POPL, 194–208 (1986).

[3] P. America, J. de Bakker, J. Kok and J. Rutten, *Denotational semantics of a parallel object-oriented language*, Information and Computation, vol.83 no.2 (1989).

[4] P. America and F. de Boer, *A Proof System for Process Creation*, Philips Research Laboratories Report RWR–116–DO–90506–DO (1990).

[5] R. Milner, **Communication and Concurrency**, Prentice-Hall (1989).

[6] R. Milner, *Functions as Processes*, Research Report 1154, INRIA (1990).

[7] R. Milner, J. Parrow and D. Walker, *A Calculus of Mobile Processes, Part I*, to appear in Information and Computation (1989).

[8] R. Milner, J. Parrow and D. Walker, *A Calculus of Mobile Processes, Part II*, to appear in Information and Computation (1989).

[9] F. Orava and J. Parrow, *Algebraic description of mobile networks: An example*, to appear in Proc. 10th IFIP Symposium on Protocol Specification, Testing and Verification, North-Holland (1990).

[10] F. Vaandrager, *Process algebra semantics for POOL*, Technical Report CS–R8629, CWI Amsterdam (1986).

[11] D. Walker, *Some results on the π-calculus*, to appear in Proc. 2nd UK/Japan Workshop on Computer Science, Oxford, September 1989, Springer LNCS.