94--001

An operational semantics and a typing system for ABCL/1 based on a calculus for objects

Vasco Thudichum Vasconcelos

April 1994

Department of Computer Science
Graduate School of Science and Technology
Keio University

3–14–1 Hiyoshi, Kohoku-ku, Yokohama 223 Japan

For information on this technical report series,
send e-mail to "yama@yy.cs.keio.ac.jp".

Abstract

The present report presents an operational semantics and a typing system for ABCL/1 programs, based on a name-passing calculus.

The particular calculus we use, TyCO, features asynchronous labelled message passing, explicit objects composed of labelled methods, and operators to compose objects and messages concurrently, and to generate new names. The calculus is only implicitly typed in that no type information is presented in programs, but else inferred through a typing assignment system.

The operational semantics is obtained by a systematic translation of the constructors present in the ABCL/1 language into TyCO processes. The type assignment system is derived from this encoding and from TyCO type inference rules.

The encoding provides a very fine grain semantics that makes explicit the order of evaluation in methods, and highlights the most complex features present in ABCL/1, namely wait-for forms and express-messages.

The typing system provides a means to statically separate "safe programs" from programs that may run into errors at runtime, and in particular avoids testing the type of arguments in methods. There is an algorithm capable of extracting the most general type of an ABCL/1 program, or to detect its inexistence.

Contents

1 Introduction **2**

2 Encoding ABCL/1 into TyCO **4**

2.1 A Fragment of ABCL/1 4
2.2 Basic Encoding 5
2.3 Simple Examples 7
2.4 Bracket Forms and Bracket Patterns 10
2.5 Selective Reception of Messages: Wait-For Forms 12
2.6 Express Messages 15
2.7 Discussion 17
2.8 Comparison With Other Encodings 18

3 Type Assignment **19**

3.1 Preliminaries 19
3.2 The Type Assignment System 20
3.3 Some Simple Examples 22
3.4 Typing Other ABCL/1 Forms 23
3.5 Principal Types and Type Inference 24
3.6 Comparison With Other Type Systems 24

1.

Introduction

ABCL/1 is a concurrent object-oriented programming language. The language itself is part of a more general project, the compilation edited by Yonezawa [9] providing a good source of information on the ABCL project.

The semantics of ABCL/1 is often given by transition systems that focus on inter-object behaviour, abstracting intra-object computations [4], similarly to semantics developed for the actor model of computation [1]. As such, the execution order of expressions and statements composing methods is understated. In addition, no formal semantics that takes care of wait-for forms and express-messages seems to be available.

The semantics we propose provides for the smallest details in the evaluation of ABCL/1 programs, from the order of evaluation of the various forms in a message send, to the implementation of complex objects containing wait-for forms and express-messages. We start with a small subset of ABCL/1 similar to ABCL$^-$ [4]. The interesting aspect of the translation of this sub-language is that to one object in ABCL/1 corresponds one main object in TyCO. To handle wait-for forms and express-messages, a more complex model is required where objects' queues have to be made explicit. This kind of objects is translated into a system of TyCO objects implementing the various components, managing the message queues and executing the object's methods.

Enqueueing messages of different kinds in a single queue poses problems in strongly typed systems. Therefore, we devise a solution in which the encoding of an ABCL/1 object comprises one different queue for each kind of message the object is able to receive. Such an organization allows to implement wait-for forms and express-messages, and could be used to fine-tune most of the object's "internals", thus providing for a precise semantics for reflection capabilities.

Based on the encoding of ABCL/1 programs, and the TyCO type inference system, we present a system of axioms and rules to assign a type to each of the constructors in the ABCL/1 language. The particular method followed ensures that well-typed programs do not run into runtime errors, and that there is an algorithm to extract the most general type of a program. Such an algorithm could be easily incorporated in an ABCL/1 interpreter or compiler thus making ABCL/1 a strongly typed language, where no type information is present in programs but else inferred by the algorithm at compilation time. Such an inference system would also give valuable information on how to correctly interact with a given program.

ABCL/1 programs are translated into TyCO, and the type system is derived from that of TyCO. At the core of TyCO lies a name-passing calculus intended to capture some basic notions present in concurrent object-oriented programming languages. The calculus features asynchronous labelled message passing, explicit objects composed of labelled methods, an operator composing objects and messages running concurrently, and a means to create new names. The reader is encouraged to scan reference [6] (in particular Section 2) to get familiarized with the calculus.

To make life easier to the programmer and the code more understandable, a number of useful derived constructors where developed [5]. These include explicit recursion, and a case statement, and will be introduced in the Chapter 2 on an "as needed" basis. The calculus is only implicitly typed in that no type information is presented in programs, but else inferred through a typing assignment system. Again, the reader is encouraged to scan Section 3 of reference [6] to have an idea of the particular form of types and the typing system used.

2.

Encoding ABCL/1 into TyCO

In this chapter we show how to encode ABCL/1 programs into TyCO, thus giving an operational semantics for ABCL/1. We start with a subset of the ABCL/1 language (described in Section 2.1), and its encoding into TyCO (Sections 2.2 and 2.3), and progressively extend the encoding to encompass more advanced features, namely bracket-forms and bracket-patterns (Section 2.4), wait-form forms (Section 2.5) and express-messages (Section 2.6). This covers most of the ABCL/1 features. Some of the remaining forms can be considered derived constructors (for example, future objects and routines can be coded into ABCL/1 itself), others can be implemented with little effort (for example loops). The last section of the chapter discusses the encoding of the remaining ABCL/1 features.

2.1 A Fragment of ABCL/1

We start with a small subset of the ABCL/1 language described by the grammar below, where x^* represents zero or more occurrences of x.

form	::=	*id*	identifiers
		var	state or temporary
		form `<==` *sendform*	now-messages
		form `<=` *sendform*	past-messages
		var `:=` *form*	assignment
		`match` *form* `(is` *pattern* *form**`)`*	match
		`object (state` *initform**`)`	
		`(script` *scriptdecl* *`)`	object definition
sendform	::=	`[`*:label* *form**`]`	
pattern	::=	`[`*:label* *id**`]`	
scriptdecl	::=	`( =>` *pattern*	
		`(temporary` *initform**`)`*form**`)`	script declaration
initform	::=	*var* `:=` *form*	initialization form

Identifiers comprise basic values such as `nil`, `0`, `1`, ..., and `+`, `-`, ..., as well as pattern-variables denoted by $x, y, \ldots$. Notice that send forms and patterns are always prefixed by some label. Unlabelled send forms or patterns are assumed to be implicitly labelled by `:val`.

ABCL/1 objects initialize their variables and execute their methods in a purely sequential fashion. In fact, except for the possibility of sending messages in parallel, there seems to be no way of expressing intra-object concurrency. As such, and since we are to encode ABCL/1 programs into a purely concurrent calculus, we need a means to sequentialize the various forms composing the objects' methods.

The sequentiality operator we are to introduce not only synchronizes the execution of two forms, but also allows to fed the result of the evaluation of the former into the latter. Since the only means to perform synchronization in TyCO is through message reception, we define the sequential operator " ; " as follows.

$$P \ ; Q \quad \stackrel{\text{def}}{=} \quad |c|(P, c \triangleright [\mathit{val}(v^*).Q]) \qquad \text{(name } c \text{ fresh)}$$

Process P, the encoding of some ABCL/1 form, upon completion of its task sends a message $\mathit{val}(v^*)$ to an agreed upon name c, where the sequence of names v^* represents the result of the evaluation of the form. Process Q, the continuation, waits for this message and goes on with its task, possibly using names v^*.

Each variable appearing in a state or temporary form will be coded into an object capable of accepting messages *read* and *write*. *read* messages come with the name intended to receive the actual value of the variable, whereas *write* messages carry not only the value to be written on the variable, but also the name of an object waiting for the completion of the operation (the continuation object). Here is the coding for ABCL/1 variables, where nil is some fixed name.

rec Var(self value) =
    self ▷ [*read*(cont). cont ◁ *val*(value), Var(self value)
        &*write*(new-value cont). cont ◁ *val*(nil), Var(self new-value)]

2.2 Basic Encoding

ABCL/1 forms are encoded into TyCO processes which, upon evaluation of the form, send a message $\mathit{val}(v^*)$, containing the result(s) v^* of the evaluation, to a continuation located at name c. The encoding function $[\![form]\!]_c$ takes an ABCL/1 *form* and a continuation name c to yield a TyCO process. Since we are more interested in the result of the evaluation of some form, rather than on the channel on which the result is communicated, we use the following notation on the definition of the encoding.

$$[\![form]\!]^{v^*} \ ; Q \quad \stackrel{\text{def}}{=} \quad |c|([\![form]\!]_c, c \triangleright [\mathit{val}(v^*).Q])$$

Notice that, in $[\![form]\!]_c$, c denotes the channel on which the result(s) of the evaluation of *form* are to be sent, whereas in $[\![form]\!]^{v^*}$, v^* denotes the result(s) of the evaluation sent along an implicit name c.

We now describe, one by one, the encoding of the various forms comprised in the grammar of Section 2.1. The result of the evaluation of an identifier is the identifier itself, and that of a variable is the value read from the variable.

$$\begin{aligned}[\![id]\!]_c &= c \lhd \textit{\textsf{val}}(id) \\ [\![var]\!]_c &= var \lhd \textit{\textsf{read}}(c)\end{aligned}$$

To evaluate a now-message, we evaluate the target-form to obtain v, then evaluate the forms to be sent to obtain v^*, and finally send the message. Notice that the result of the now-message will be sent directly to the continuation.

$$[\![form \texttt{ <== [}{:}label\ form^* \texttt{]}]\!]_c \quad = \quad [\![form]\!]^v ;\ [\![\texttt{[}form^*\texttt{]}]\!]^{v^*} ;\ v \lhd label(v^* c)$$

Past-messages are handled similarly, with the difference that the result of the evaluation is simply name nil.

$$[\![form \texttt{ <= [}{:}label\ form^* \texttt{]}]\!]_c \quad = \quad [\![form]\!]^v ;\ [\![\texttt{[}form^*\texttt{]}]\!]^{v^*} ;\ (v \lhd label(v^*), c \lhd \textit{\textsf{val}}(\textsf{nil}))$$

Send-forms are evaluated from left to right. Their results are accumulated, and the continuation receives the sequence of names representing the result of evaluating the forms in the send-form.

$$\begin{aligned}[\![\texttt{[]}]\!]_c &= c \lhd \textit{\textsf{val}}() \\ [\![\texttt{[}form\ form^*\texttt{]}]\!]_c &= [\![form]\!]^v ;\ [\![\texttt{[}form^*\texttt{]}]\!]^{v^*} ;\ c \lhd \textit{\textsf{val}}(v v^*)\end{aligned}$$

To evaluate an *assign form* we evaluate the right-hand side and send the variable to be updated a ***read*** message with the computed value. Notice that the continuation is activated directly by the variable's ***write*** method.

$$[\![var \texttt{ := } form]\!]_c \quad = \quad [\![form]\!]^v ;\ var \lhd \textit{\textsf{write}}(vc)$$

To evaluate a *match form* we first evaluate the form to be matched, and branch according to the result. This branching is performed by an object whose method's heads are precisely the patterns in the match form, and whose bodies are the encodings of the forms in the match form.

$$\begin{aligned}&[\![\texttt{match } form \texttt{ (is [}{:}label_1\ x_1^*\texttt{]}\ form_1^*\texttt{)} \cdots \texttt{(is [}{:}label_n\ x_n^*\texttt{)}\ form_n^*\texttt{]}]\!]_c \quad = \\ &\qquad |c'|([\![form]\!]_{c'}, c' \rhd [label_1(x_1^*).[\![form_1^*]\!]_c \& \ldots \& label_n(x_n^*).[\![form_n^*]\!]_c])\end{aligned}$$

Sequences of forms are evaluated similarly to send-forms above, with the difference that we only remember the value of the last evaluated form.

$$\begin{aligned}[\![\,]\!]_c &= c \lhd \textit{\textsf{val}}(\textsf{nil}) \\ [\![form^*\ form]\!]_c &= [\![form^*]\!] ;\ [\![form]\!]_c\end{aligned}$$

To evaluate an *initialization form*, we first evaluate the right-hand side form, and then create a new variable with the result of the evaluation. The location (or the address) of the variable is a newly created name that will be transmitted to the continuation as the result of the evaluation of the initialization form.

$$[\![var \texttt{ := } form]\!]_c \quad = \quad [\![form]\!]^v \ ; \ |\mathsf{var}|(\mathsf{Var}(\mathsf{var}\ v), c \triangleleft \mathit{val}(\mathsf{var}))$$

Sequences of initialization forms are handled in much the same way as send-forms. To evaluate the *body of a script*, we evaluate the initialization forms of the temporary variables, and then the sequence of forms.[1]

$$[\![\texttt{(temporary } initform^* \texttt{)}\ form^*]\!]_c \quad = \quad [\![initform^*]\!]^{v^*} \ ; \ [\![form^*]\!]_c$$

Finally, we handle ABCL/1 *objects*. To each ABCL/1 object corresponds one recursive TyCO object, implementing the methods of the ABCL/1 object, plus one object for each local variable of the ABCL/1 object. Initialization forms are evaluated first. Then we create a recursive object whose parameters are the result of the evaluation of the initialization forms (that is the object's `state` variables), together with a newly created name: the location (or address) of the object. The result of evaluating an object form is exactly this newly created name.[2]

$$\begin{array}{l} [\![\texttt{[object (state } initform^* \texttt{)} \\ \qquad\qquad \texttt{[script (=> [}{:}label_1\ x_1^* \texttt{]}\ body_1 \texttt{)} \cdots \texttt{(=> [}{:}label_n\ x_n^* \texttt{]}\ body_n \texttt{))]}]\!]_c = \\ \textbf{letrec}\ \mathsf{Obj}(\mathsf{me}\ v^*) = \\ \qquad \mathsf{me} \triangleright [label_1(x_1^*).[\![body_1]\!] \ ; \ \mathsf{Obj}(\mathsf{me}\ v^*) \& \cdots \& label_n(x_n^*).[\![body_n]\!] \ ; \ \mathsf{Obj}(\mathsf{me}\ v^*)] \\ \textbf{in}\ [\![initform^*]\!]^{v^*} \ ; \ |\mathsf{obj}|(\mathsf{Obj}(\mathsf{obj}\ v^*), c \triangleleft \mathit{val}(\mathsf{obj})) \end{array}$$

The new constructors can be easily translated into core-TyCO. Recursive definitions are translated into replication as shown in reference [6]. A process of the form **let** $X(x^*) = P$ **in** Q behaves as Q with occurrences of $X(v^*)$ replaced by $P[v^*/x^*]$, that is, by P with occurrences of names x^* replaced by v^*.

2.3 Simple Examples

This section presents examples of the encoding of some simple ABCL/1 objects taken from the book on ABCL [9].

Our first example is the `greeting` object on page 239, which, by expanding the !-notation, can be written as follows.

```
[object greeting
  (script
    (=> [:hello reply-to]
        reply-to <= :hi))]
```

[1] To avoid renaming variable-names in *form** we assume that v^* is the sequence of `temporary` variables initialized in *initform**.

[2] Again we assume that the names in v^* are exactly the `state` variables in *initform**.

We begin by encoding the past-message form. The evaluation of the target object with continuation c_1 is $c_1 \lhd \mathit{val}(\text{reply-to})$. Then, the encoding of the whole message with continuation c_2 becomes the following.

$$|c_1|(\ c_1 \lhd \mathit{val}(\text{reply-to}),\ c_1 \rhd [\mathit{val}(\text{reply-to}).\ \text{reply-to} \lhd \mathit{hi},\ c_2 \lhd \mathit{val}(\text{nil})])$$

The body of the script declaration is encoded as follows, where Obj is a recursive call to the greeting object, to be bound later.

$$\begin{array}{l} |c_2|(|c_1|(c_1 \lhd \mathit{val}(\text{reply-to}), \\ \qquad\quad c_1 \rhd [\mathit{val}(\text{reply-to}).\ \text{reply-to} \lhd \mathit{hi},\ c_2 \lhd \mathit{val}(\text{nil})]), \\ \qquad c_2 \rhd [\mathit{val}(\_).\ \text{Obj}(\text{me})]) \end{array}$$

The encoding, although correct is not much efficient. An intelligent compiler could short-cut the unnecessary synchronizations, to obtain a more efficient and readable program.

$$\text{reply-to} \lhd \mathit{hi},\ \text{Obj}(\text{me})$$

In this way, the encoding of the greeting object with continuation c, after some partial evaluation, becomes the following.

$$\begin{array}{l} \textbf{letrec}\ \text{Obj}(\text{me}) = \text{me} \rhd [\mathit{hello}(\text{reply-to}).\ \text{reply-to} \lhd \mathit{hi},\ \text{Obj}(\text{me})] \\ \textbf{in}\ |\text{greeting}|(\text{Obj}(\text{greeting}),\ c \lhd \mathit{val}(\text{greeting})) \end{array}$$

For a more complex, yet quite simple example, we present the encoding for the `counter` object on page 239, which can be written as follows.

```
[object counter
  (state [count := 0]
  (script
    (=> :increment
        [count := [succ <== count]])
    (=> [:value reply-to]
        reply-to <= count)))]
```

We start be encoding the initialization form. The right-hand side, 0, is evaluated to itself. The initialization form with continuation c_1 (and after short-cutting the continuation of the evaluation of 0) becomes $|\text{var}|(\text{Var}(\text{var } 0),\ c_1 \lhd \mathit{val}(\text{var}))$.

The past-message in method value is similar to the one in the greeting object, only that now we need to read count first. The body of the method, after short-cutting the continuation of the evaluation of reply-to, becomes

$$\begin{array}{l} |c_2|(\text{count} \lhd \mathit{read}(c_2), \\ \quad c_2 \rhd [\mathit{val}(\text{count-value}).\ \text{reply-to} \lhd \mathit{val}(\text{count-value}),\ \text{Obj}(\text{me count})]) \end{array}$$

Notice that we cannot short-cut continuation c_2, since object Obj could recur (and hence accept new messages and increment count) before count being actually read.

We then turn our attention to method increment, and start by encoding the right-hand side of the assignment. We need to evaluate the identifier succ and the variable count. Identifier succ evaluates into itself; variable count evaluates into a message count $\lhd$*read*(c_3). The code to evaluate the now-message with continuation c_4 is as follows.

$$
\begin{array}{l}
|\mathsf{c}_3|(\mathsf{count} \lhd \mathit{read}(\mathsf{c}_3), \\
\quad \mathsf{c}_3 \rhd [\mathit{val}(\mathsf{count\text{-}value}).\ \mathsf{succ} \lhd \mathit{val}(\mathsf{count\text{-}value}\ \mathsf{c}_4)])
\end{array}
$$

Then, the assignment form with continuation c_5 becomes

$$
\begin{array}{l}
|\mathsf{c}_4|(|\mathsf{c}_3|(\mathsf{count} \lhd \mathit{read}(\mathsf{c}_3), \\
\qquad \mathsf{c}_3 \rhd [\mathit{val}(\mathsf{count\text{-}value}).\ \mathsf{succ} \lhd \mathit{val}(\mathsf{count\text{-}value}\ \mathsf{c}_4)]), \\
\quad \mathsf{c}_4 \rhd [\mathit{val}(\mathsf{new\text{-}value}).\ \mathsf{count} \lhd \mathit{write}(\mathsf{new\text{-}value}\ \mathsf{c}_5)])
\end{array}
$$

Putting everything together, the (partially evaluated) encoding of the counter object with continuation c is as follows.

$$
\begin{array}{l}
\textbf{letrec}\ \mathsf{Obj}(\mathsf{me}\ \mathsf{count}) = \\
\qquad \mathsf{me} \rhd [\ \mathit{increment}. \\
\qquad\qquad |\mathsf{c}_5 \mathsf{c}_4\ \mathsf{c}_3|(\\
\qquad\qquad\quad \mathsf{count} \lhd \mathit{read}(\mathsf{c}_3), \\
\qquad\qquad\quad \mathsf{c}_3 \rhd [\mathit{val}(\mathsf{count\text{-}value}).\ \mathsf{succ} \lhd \mathit{val}(\mathsf{count\text{-}value}\ \mathsf{c}_4)], \\
\qquad\qquad\quad \mathsf{c}_4 \rhd [\mathit{val}(\mathsf{new\text{-}value}).\ \mathsf{count} \lhd \mathit{write}(\mathsf{new\text{-}value}\ \mathsf{c}_5)], \\
\qquad\qquad\quad \mathsf{c}_5 \rhd [\mathit{val}(\_).\ \mathsf{Obj}(\mathsf{me}\ \mathsf{count})]) \\
\qquad\quad \&\, \mathit{value}(\mathsf{reply\text{-}to}). \\
\qquad\qquad |\mathsf{c}_2|(\mathsf{count} \lhd \mathit{read}(\mathsf{c}_2), \\
\qquad\qquad\quad \mathsf{c}_2 \rhd [\mathit{val}(\mathsf{count\text{-}value}).\ \mathsf{reply\text{-}to} \lhd \mathit{val}(\mathsf{count\text{-}value}), \\
\qquad\qquad\qquad\qquad \mathsf{Obj}(\mathsf{me}\ \mathsf{count})])] \\
\textbf{in}\ |\mathsf{var}\ \mathsf{counter}|(\mathsf{Var}(\mathsf{var}\ 0),\ \mathsf{Obj}(\mathsf{counter}\ \mathsf{var}),\ \mathsf{c} \lhd \mathit{val}(\mathsf{counter}))
\end{array}
$$

In simple cases, when state variables are never written, or when the assignments to *all* variables can be postponed until the point where the object recurs, we can dispense variables (and hence reading and writing them) and simply recur with the new value of the variables.[3] The simplified code for the counter object should be self-explanatory.

[3]Cf. the "become" constructor of the actor model.

$$
\begin{array}{l}
\textbf{letrec}\ \mathsf{Obj(me\ count)} = \\
\qquad\qquad \mathsf{me} \triangleright [\mathit{increment}. \\
\qquad\qquad\qquad\qquad |\mathsf{c_1}|(\mathsf{succ} \lhd \mathit{val}(\mathsf{count\ c_1}), \\
\qquad\qquad\qquad\qquad\quad \mathsf{c_1} \triangleright [\mathit{val}(\mathsf{new\text{-}count}).\ \mathsf{Obj(me\ new\text{-}count)}]) \\
\qquad\qquad\quad \&\mathit{value}(\mathsf{reply\text{-}to}). \\
\qquad\qquad\qquad\qquad \mathsf{reply\text{-}to} \lhd \mathit{val}(\mathsf{count}),\ \mathsf{Obj(me\ count)}] \\
\textbf{in}\ |\mathsf{counter}|(\mathsf{Obj(counter\ 0)},\ \mathsf{c} \lhd \mathit{val}(\mathsf{counter}))
\end{array}
$$

2.4 Bracket Forms and Bracket Patterns

The basic grammar in Section 2.1 does not contemplate bracket-forms. In particular, the contents of a message is not really a list but else a fixed-length sequence of names. As a result some flexibility is lost, particularly when we would like to express methods handling arbitrary long messages, or when lists of names as target of messages are used for multicast.

We extend the syntax of ABCL/1 in Section 2.1 by introducing the following production.

$$
\mathit{form} \quad ::= \quad \cdots \quad | \quad [\mathit{form}.\ \mathit{form}]
$$

More complex, derived bracket-forms can be obtained as below.

$$
\begin{array}{rcl}
[\mathit{form}_1 \cdots \mathit{form}_n .\ \mathit{form}] & \stackrel{\text{def}}{=} & [\mathit{form}_1 . \cdots [\mathit{form}_n .\ \mathit{form}] \cdots] \\
[\mathit{form}_1 \cdots \mathit{form}_n] & \stackrel{\text{def}}{=} & [\mathit{form}_1 \cdots \mathit{form}_n .\ \mathsf{nil}]
\end{array}
$$

Bracket-forms are encoded as lists. Different kinds of lists are needed for different kinds of bracket-forms. Simple bracket-forms, those appearing anywhere but as target of messages, are encoded with simple lists. Past and now multicast need special purpose lists.

Simple lists can be built in TyCO from two constructors, Nil and Cons, accepting a single message *val*. The former replies *nil*, stating "I am a nil cell"; whereas the latter replies *cons*(head tail) stating "I am a cons cell and here is my first and second elements."

$$
\textbf{rec}\ \mathsf{Nil(self)} = \mathsf{self} \triangleright [\mathit{val}(\mathsf{reply\text{-}to}).\ \mathsf{reply\text{-}to} \lhd \mathit{nil},\ \mathsf{Nil(self)}]
$$

$$
\begin{array}{l}
\textbf{rec}\ \mathsf{Cons(self\ head\ tail)} = \mathsf{self} \triangleright [\mathit{val}(\mathsf{reply\text{-}to}).\ \mathsf{reply\text{-}to} \lhd \mathit{cons}(\mathsf{head\ tail}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Cons(self\ head\ tail)}]
\end{array}
$$

Then, bracket-forms occurring anywhere *but as targets of messages* can naturally be coded as below.

$$
[\![\, [\mathit{form}.\ \mathit{form}] \,]\!]_c \quad = \quad [\![\mathit{form}]\!]^h\ ;\ [\![\mathit{form}]\!]^t\ ;\ |l|(\mathsf{Cons}(lht), c \lhd \mathit{val}(l))
$$

As an example, the body `stack := [x.stack]` of the push method in the stack object ([9], page 264) can be coded, with continuation c, as follows.

```
[[stack := [x.stack]]]_c =
[[[x.stack]]]^new-stack ; stack ◁ write(new-stack c) =
|c'|(stack ◁ read(c'),
    c' ▷ [val(stack-value). |new-stack|(Cons(new-stack x stack-value),
                                        stack ◁ write(new-stack c)])
```

To handle bracket-patterns we extend the syntax of patterns as follows.

$$\begin{array}{lcl} pattern & ::= & \cdots \;\;|\;\; \texttt{[:}label\ brktpat\texttt{]} \\ brktpat & ::= & id \;\;|\;\; \texttt{[}brktpat\texttt{.}\ brktpat\texttt{]} \end{array}$$

The compilation of patterns is a well studied subject, and the technique described by Peyton Jones [2] could be used to compile ABCL/1 patterns into TyCO code. As a simple example, suppose we have the following object defining the **null** function.

```
[object nullCheck
  (script
    ( => [:null?  []] !  :true)
    ( => [:null?  [_._]] !  :false))
```

The two methods can be encoded into the following TyCO method.

```
null?(x reply-to). |c|(x ◁val(c),
                      c ▷ [ nil. reply-to ◁ true
                           &cons(_ _). reply-to ◁ false]
```

Lists for *past-multicast* (that is lists used in the code of the left hand-side of a past-message) must be of a special kind. Suppose such a list is to receive multicast messages of the form `[:label x]`. Then, lists for past-multicast are built from cells PastCons and PastNil, obtained from the corresponding cells for simple lists, by adding a new method *label*(x), as described below. In this way message `[:label x]` is broadcast to all the names in the list.

```
rec PastNil(self) = self ▷ [ val ···
                            &label(_). PastNil(self)]

rec PastCons(self head tail) = self ▷ [ val ···
                                       &label(x). head ◁ label(x), tail ◁ label(x),
                                                  PastCons(self head tail)]
```

The encoding of bracket-forms at the target of messages is exactly the same as for simple bracket-forms, only that the list constructors to be used are PastNil and PastCons, instead of Nil and Cons.

List constructors for *now-multicast* (that is lists in the code of the right-hand side of a now-message) are similar to those of past-multicast, with the difference that they have to wait for the answers to the multicast messages and build a list of replies. The actual code for NowNil and NowCons is left as an exercise.

2.5 Selective Reception of Messages: Wait-For Forms

So far, all ABCL/1 forms we considered required a very simple model of objects. In particular, there was no need to explicitly enqueue incoming messages since the communication system underlying the execution of TyCO programs holds all messages (and objects) until interaction is possible. As a result, to each ABCL/1 object there corresponded a single TyCO object (plus those implementing the `state` variables).

The semantics of wait-for forms and express-messages cannot be captured by such a simplistic model. We therefore develop a model assigning each ABCL/1 object containing wait-for forms a collection of interacting TyCO objects described below.

- A *receptor* always ready to accept all messages (in top-level and wait-for script declarations) the ABCL/1 object is capable of receiving.
- One *queue* for each kind of message the ABCL/1 object is capable of receiving, and always ready to enqueue messages sent by the receptor.
- One *top-level server* selecting and dequeuing messages from the top-level queues, and waiting if all queues are empty.
- One *wait server* for each wait-form existing in the ABCL/1 object, selecting and dequeuing messages from the corresponding wait-for queues, and waiting if all queues are empty.
- One *executor* accepting messages from the servers (both top-level and wait), executing the body corresponding to that in the ABCL/1 object, and activating the appropriate server upon completion.

Let us illustrate the encoding through a slight simplification of the window-controller example ([9], page 262).

```
[object window-controller
  (state [window := [create-window <== [:new]]])
  (script
    (=> [:write character]
        [window <= [:write character]])
    (=> [:lock]
        !:locked
        (wait-for
          (=> [:release]
              !:unlocked))))]
```

The receptor maintains a different queue for each kind of message the window-controller is able to receive. These include the top-level messages *write* and *lock*, as well as the wait-for message *release*. All the receiver has to do is to enqueue incoming messages in the appropriate queues.

```
rec Receptor(self writeQueue lockQueue releaseQueue) =
   self ▷ [ write(character).
                 writeQueue ◁ enqueue(character),
                 Receptor(self writeQueue lockQueue releaseQueue)
            & lock(locker).
                 lockQueue ◁ enqueue(locker),
                 Receptor(self writeQueue lockQueue releaseQueue)
            & release(unlocker).
                 releaseQueue ◁ enqueue(unlocker),
                 Receptor(self writeQueue lockQueue releaseQueue)]
```

Each queue maintains a list elems of the elements enqueued and the name wait of an object eventually waiting for the queue not to be empty (typically a server), and provides three operations: *enqueue*, *dequeueWithCont* and *ready*. The *enqueue*(x) method of the write queue places x in list elems and sends wait a *writeReady* message, just in case object wait is still waiting for the queue not to be empty. The *dequeueWithCont* method replies the head of the list together with the specified continuation. Finally, the *ready* method replies *writeReady* or lets the sender wait, depending on the status of elems.[4]

```
rec WriteQueue(self elems wait) =
   self ▷ [ enqueue(x).
                 wait ◁ writeReady,
                 WriteQueue(self cons(x elems) wait)
            & ready(reply-to).
                 case elems ◁ val
                 of [ nil. WriteQueue(self elems reply-to)
                    & cons(_ _). reply-to ◁ writeReady, WriteQueue(self elems wait)]
            & dequeueWithCont(reply-to cont).
                 case elems ◁ val
                 of [ nil. WriteQueue(self elems wait)
                    & cons(head tail). reply-to ◁ write(head cont),
                                       WriteQueue(self tail wait)]]
```

The new constructor introduced above, WriteQueue(self *cons*(x elems) wait) means |elems'|(Cons(elems' x elems), WriteQueue(self elems' wait)) [5].

The code for both the top-level and the wait-for server follows the same template. Each server interacts with the corresponding queues and the executor, and orchestrates all the continuations, ensuring that methods are executed in mutual exclusion. Servers have a single method, *go*, with no arguments. The top-level server of the window-controller object, upon reception of a message *go*, enquires the state of its queues and accepts the first reply (and only this) in the form of a message *writeReady* or *lockReady*. It then dequeues the corresponding queue directly to the

[4]The code actually implements a FILO policy, rather than a FIFO.

executor, specifying the continuation server. Methods like *write* are sent the name of the top-level server as the continuation; methods like *lock* which terminate with a wait-for form, are sent the wait server as the continuation. The code for the top-level server of the window-controller is depicted below; the one for the wait server is similar.

```
rec ToplevelServer(self executor waitServer writeQueue lockQueue) =
  self ▷ [ go. case writeQueue ◁ ready, lockQueue ◁ ready
                of [ writeReady.
                        writeQueue ◁ dequeueWithCont(executor self),
                        ToplevelServer(self executor waitServer writeQueue lockQueue)
                   & lockReady.
                        lockQueue ◁ dequeueWithCont(executor waitServer),
                        ToplevelServer(self executor waitServer writeQueue lockQueue)]]
```

Notice that in case both queues are empty, the top-level server blocks and waits for a message to arrive at one of the queues. Notice also that the strategy of pooling the queues and dequeuing one of them only works in a concurrency setting because we are guaranteed to have at most one server active at a time.

The executor has a method for each script declaration in the ABCL/1 object, both those at the top-level and those in wait-for forms. It executes the corresponding method and activates, through a message *go*, the supplied continuation. The optimized code for the executor is depicted below.

```
rec Executor(self window) =
  self ▷ [ write(character cont).
                window ◁ write(character),
                cont ◁ go,
                Executor(self window)
         & lock(locker cont).
                locker ◁ locked,
                cont ◁ go,
                Executor(self window)
         & release(unlocker cont).
                unlocker ◁ unlocked,
                cont ◁ go,
                Executor(self window)]
```

Finally, by putting everything together, we obtain the following encoding (with continuation c) for the window-controller object.

|receptor writeQueue lockQueue releaseQueue|(*The receptor and the queues*
    Receptor(receptor writeQueue lockQueue releaseQueue),
    WriteQueue(writeQueue *nil* _),
    LockQueue(lockQueue *nil* _),
    ReleaseQueue(releaseQueue *nil* _),
|toplevelServer waitServer executor|(*The two servers*
    ToplevelServer(toplevelServer executor waitServer writeQueue lockQueue),
    WaitServer(waitServer executor toplevelServer releaseQueue),
    Executor(executor someWindow), *The executor*
    toplevelServer $\lhd$ *go*, *Activate the top-level server*
    c $\lhd$ *val*(receptor))) *The result of the evaluation*

Similarly to what we did in the definition of WriteQueue, the expression WriteQueue(writeQueue *nil* _) means |n x|(Nil(n), WriteQueue(writeQueue n x)) [5].

2.6 Express Messages

The model developed in the previous section easily accommodates the management of express-messages. The idea is to place, at appropriate points in the code of methods, calls to an expressChecker object. Such an object pools all express-queues (that is queues associated with express-messages) and activates an express method if one of the express queues is not empty. If no express-message is waiting, the expressChecker reactivates the execution of the method, otherwise the express-method selected directly reactivates, upon completion, the execution of the interrupted method. In this way we ensure that the computation of normal methods progresses, even in the presence of an heavy load of express-messages.

Let us see how to encode an object containing express-messages through the clock example ([9], page 256).

```
[object clock
  (state [time:=0])
  (script
    (=> [:start]
        (loop
          (sleep 1)
          [time := (1+ time)]))
    (=>>[:what-time]
        !time))]
```

For the expressChecker object to be able to pool the express-queues without risking suspension when all queues are empty, express-queues need a more conventional dequeue operation. In this way, express-queues are obtained from queues in Section 2.5 by adding the following method.

```
dequeue(reply-to).
  case elems ◁ val
  of [ nil. reply-to ◁ empty, WhatTimeQueue(self elems wait)
    &cons(head tail). reply-to ◁ dequeued(head), WhatTimeQueue(self tail wait)]
```

The object to check the existence of express-messages and execute one of them can be coded as follows. Notice the usage of the continuation: passed to the executor if the express-queue is not empty, and directly activated otherwise.

```
rec ExpressChecker(self executor whatTimeQueue) =
  self ▷ [ val(cont).
           case whatTimeQueue ◁ dequeue
           of [ dequeued(reply-to).
                  executor ◁ what-time(reply-to cont),
                  ExpressChecker(self executor whatTimeQueue)
              &empty.
                  cont ◁ val,
                  ExpressChecker(self executor whatTimeQueue)]]
```

The Executor is encoded in much the same way as before, only that now, between every two forms in a sequence of forms, we check the existence of pending express-messages. This can be achieved by placing a message expressChecker ◁ *val*(cont) in between the synchronization code for two forms, where cont is the location of the second form, that is, by simply replacing P ; Q by

$$P \;;\; \mathsf{expressChecker} \;;\; Q.$$

Finally, by putting everything together, the original clock object is encoded (with continuation c) as follows.

```
|receptor startQueue whatTimeQueue|(     The receptor and the queues
    Receptor(receptor startQueue whatTimeQueue),
    startQueue(startQueue nil _),
    whatTimeQueue(whatTimeQueue nil _),
|server expressChecker executor|(         The server and the expressChecker
    ToplevelServer(server executor startQueue whatTimeQueue),
    ExpressChecker(expressChecker executor whatTimeQueue),
    Executor(executor expressChecker),   The executor
    toplevelServer ◁ go,                 Activate the top-level server
    c ◁ val(receptor)))                  The result of the evaluation
```

Atomic forms are quite easy to implement: it suffices to refrain from calling the expressChecker within these forms. Also, non-resume forms pose no particular problem. The body of a method containing a non-resume form should call the toplevelServer instead of the supplied continuation. For example, method

```
(=>> [:stop]
     (non-resume))
```

of the `clock` object ([9], page 258) can be coded into the following method of the Executor.

$$\begin{array}{l} \textit{stop}(\mathsf{continuation}). \\ \quad \mathsf{toplevelServer} \lhd \textit{go}, \\ \quad \mathsf{Executor(self\ toplevelServer\ expressChecker\ time)} \end{array}$$

2.7 Discussion

The proposed encoding does not follow the semantics of ABCL/1 in (at least) two aspects.

1. It does not evaluate the state variables of an object only after the reception of the first message, and
2. it does not implement the "transition ordering law."

The first of the above faults can be easily fixed. All we have to do is to provide each object with a front-end object capable of receiving exactly the type of messages the object is capable of. Upon reception of a message, the front-end instantiates a copy of the object (thus evaluating the local variables) and forwards the message to the new object.

Suppose the ABCL/1 object to be encoded has methods with patterns [$:label_1\ x_1^*$], ..., [$:label_n\ x_n^*$]. The encoding of the object, with lazy evaluation of state variables and with continuation c is as follows.

$$\begin{array}{l} |\mathsf{obj}|(\mathsf{obj} \rhd [\ label_1(x_1^*).\ [\![object]\!]^{\mathsf{obj}}\ ;\ \mathsf{obj} \lhd label_1(x_1^*) \\ \qquad\qquad\quad \vdots \\ \qquad\qquad\quad \& label_n(x_n^*).\ [\![object]\!]^{\mathsf{obj}}\ ;\ \mathsf{obj} \lhd label_n(x_n^*)] \\ \quad \mathsf{c} \lhd \textit{val}(\mathsf{obj})) \end{array}$$

The transition ordering law is somewhat more complicated to implement in a calculus that makes no assumption on the order of reception of messages. We can tag each message with a pair (target-object-name, unique-number) at sending time and order the messages at the receptor. While tagging messages at the sender can be reasonably simple to implement, ordering messages at the receptor introduces some complications. In particular, the model with a distinct message queue for each kind of message, introduced to handle wait-for forms and express-messages, does not seem much adequate to order messages since these are distributed among several queues.

We conclude this section with a brief overview of those features in ABCL/1 we did not handle. Loops and conditionals are easy to implement, either as ABCL/1 derived forms, or by encoding into TyCO. Reply-destination variables, routines, reply-forms and future objects can be encoded in ABCL/1 itself. There are two kinds of constraints in script declarations: type and value-checking. The former kind can be completely eliminated by a static type-checking of ABCL/1 programs (see the next chapter). The latter kind requires the generation of some special code to test the condition. The implementation of past-type parallel-message passing should be quite simple. As for the now-type, we can make use of the special purpose list cells NowNil and NowCons described in Section 2.4, to collect the replies to the messages. All that remains is to encode Lisp programs.

2.8 Comparison With Other Encodings

Walker showed how to describe a semantics for concurrent object-oriented programming languages by a systematic translation of the constructors in a POOL-like language into the π-calculus [7].

While it is possible to encode TyCO into the π-calculus (and vice-versa) [6], labelled sums yield a much more elegant description of methods, with a direct counterpart in the type system. The cited work assigns each method in an object a different name, and multiplex all these names into a single name. This makes the invocation of a method a tree-way protocol, against a one-way in TyCO.

We followed a systematic approach to encode ABCL/1 programs into TyCO processes, where each ABCL/1 form is encoded into a process producing a message *val*(v), for v the result of the evaluation of the form. The encoding is quite flexible and modular, and easily extends to accommodates more complex constructs. In contrast to Walker's work, the TyCO code produced, albeit in a "concurrent assembly language", is still understandable.

3.

Type Assignment

Although we could type check ABCL/1 programs by translating these into TyCO, and run TyCO's type checker on the obtained code, we can derive a type inference system to check ABCL/1 source code directly. This chapter presents a type assignment system for ABCL/1 programs. In particular, Section 3.1 presents the form of the statements of the typing system and discusses some properties of this. Then, Section 3.2 introduces the axioms and rules of the system for the fragment of ABCL/1 discussed in Section 2.1. Section 3.3 discusses types for the examples considered in Section 2.3. Section 3.4 deals with some of the remaining forms in ABCL/1, and the last section of the chapter discusses the existence of principal typings and a typing inference algorithm.

3.1 Preliminaries

From the encoding of ABCL/1 forms proposed in Chapter 2 and TyCO's typing assignment system we deduce a typing system for ABCL/1 programs.

Types in TyCO are built from a set of type-variables (and possibly some constant types) by means of a single constructor: a record. Types are inductively defined by the following grammar

$$\alpha \quad ::= \quad t \quad | \quad [l_1 \colon \alpha_1^*, \ldots l_n \colon \alpha_n^*]$$

where t represents a type variable, α a type, and α^* a sequence of zero or more types, where $n \geq 0$. The reader is encouraged to scan Section 3 of reference [6] to get an idea of the TyCO type inference system.

If Γ is a basis (that is a collection of name-type pairs) and α is the type of the name *form* evaluates to, we write

$$\Gamma \vdash_{\mathrm{abcl}} \mathit{form} \triangleright \alpha$$

if we can prove, starting from basis Γ and using the system of axioms and rules described in the next section, that *form* has a type α. In this case, we say *form* is well-typed. The typing assignment system for ABCL/1 satisfies the following proposition.

$$\Gamma \vdash_{\text{abcl}} \mathit{form} \triangleright \alpha \qquad \text{if} \qquad \vdash_{\text{tyco}} [\![\mathit{form}]\!]_c \triangleright \Gamma \cdot c\colon [\mathsf{\textit{val}}\colon \alpha]$$

The simple fact that each of the typing rules for ABCL/1 satisfy the above proposition, enables the ABCL/1 typing system to inherit from TyCO a number of most desirable properties, including the ones below.

- If we can deduce $\Gamma \vdash_{\text{abcl}} \mathit{form} \triangleright \alpha$, then all the free names occurring in *form* are in Γ.

- If an ABCL/1 program is well-typed, then no object will ever receive a message for which it hasn't an appropriate method (this includes the right label and the right number and type of arguments).

- There is a principal typing for each well-typed ABCL/1 program, from which all typings that make the program well-typed can be obtained. Furthermore such a typing can be effectively computed.

We start by investigating the type of ABCL/1 variables. Remember from Section 2.1 that variables are coded into objects capable of receiving two kinds of messages: *read* and *write*. With *read* comes a name capable of receiving a message *val*(v), where v is the value the variable is holding. With *write* comes a pair of names: a value and a name capable of receiving messages of the form *val*(nil). By using the typing rules of TyCO, it is not difficult to prove that a type for a name standing for an ABCL/1 variable is

$$[\mathsf{\textit{read}}\colon [\mathsf{\textit{val}}\colon \alpha],\ \mathsf{\textit{write}}\colon \alpha\ [\mathsf{\textit{val}}\colon \mathsf{Nil}]]$$

where α is the type of the value the variable is holding and Nil is the type of name nil. Such a type will be denoted by $\mathrm{Var}(\alpha)$ in the sequel.

3.2 The Type Assignment System

This section presents the system of rules and axioms for the fragment of ABCL/1 considered in Section 2.1. Each of the rules described below can be easily obtained from the corresponding encoding and a small deduction in TyCO's type assignment system. The notation $\Gamma \cdot x\colon \alpha$ means $\Gamma \cup \{x\colon \alpha\}$, provided x does not appear in Γ.

The result of the evaluation of an identifier is the identifier itself, and that of a variable is the value stored in the variable.

(Id) $$\Gamma \vdash \mathit{id} \triangleright \alpha \qquad (\mathit{id}\colon \alpha \in \Gamma)$$

(Var) $$\Gamma \vdash \mathit{var} \triangleright \alpha \qquad (\mathit{var}\colon \mathrm{Var}(\alpha) \in \Gamma)$$

If the forms to be sent in a past-message have types α^*, and the target is ready to accept a message of type label: α^*, then the type of a past-message send is the nil type.

$$(\textsc{Past-msg}) \qquad \frac{\Gamma \vdash \mathit{form} \triangleright [\mathit{label}\colon \alpha^*, \ldots] \qquad \Gamma \vdash \texttt{[}\mathit{form}^*\texttt{]} \triangleright \alpha^*}{\Gamma \vdash \mathit{form} \texttt{ <= } [\mathit{:label}\ \mathit{form}^*] \triangleright \mathrm{Nil}}$$

The case of now-messages is similar, with the difference that the target must be ready to accept a message of type label: $\alpha^*[\mathit{val}\colon \beta]$, thus replying a message of type val: β. In this case, β is the type of the now-message send.

$$(\textsc{Now-msg}) \qquad \frac{\Gamma \vdash \mathit{form} \triangleright [\mathit{label}\colon \alpha^*[\mathit{val}\colon \beta], \ldots] \qquad \Gamma \vdash \texttt{[}\mathit{form}^*\texttt{]} \triangleright \alpha^*}{\Gamma \vdash \mathit{form} \texttt{ <== } \texttt{[}\mathit{:label}\ \mathit{form}^*\texttt{]} \triangleright \beta}$$

The type of a sequence of forms to be sent is just the juxtaposition of the types of the individual forms, where ε is the empty sequence of types.

$$(\textsc{Send-empty}) \qquad \Gamma \vdash \texttt{[]} \triangleright \varepsilon$$

$$(\textsc{Send-seq}) \qquad \frac{\Gamma \vdash \mathit{form} \triangleright \alpha \qquad \Gamma \vdash \texttt{[}\mathit{form}^*\texttt{]} \triangleright \beta^*}{\Gamma \vdash \texttt{[}\mathit{form}\ \mathit{form}^*\texttt{]} \triangleright \alpha\beta^*}$$

If *form* is of type α and *var* is of type $\mathrm{Var}(\alpha)$, then the type of the assignment *var* := *form* is the Nil type.

$$(\textsc{Assign}) \qquad \frac{\Gamma \vdash \mathit{form} \triangleright \alpha}{\Gamma \vdash \mathit{var} \texttt{ := } \mathit{form} \triangleright \mathrm{Nil}} \qquad (\mathit{var}\colon \mathrm{Var}(\alpha) \in \Gamma)$$

If the evaluation of *form* can result in a message of type label: α^* or of type label': β^*, if the pattern variables x^* in `(is` $\texttt{[}\mathit{:label}\ x^*\texttt{]}\ \mathit{form}^*$`)` are of types α^* (and similarly for y^*), and if form^* and form'^* are of the same type γ, then γ is the type of the match form. Notice that we discharge assumptions on the pattern variables, since these are local to the match.

(Match)

$$\frac{\Gamma \vdash \mathit{form} \triangleright [\mathit{label}\colon \alpha^*, \mathit{label}'\colon \beta^*] \quad \Gamma \cdot x^*\colon \alpha^* \vdash \mathit{form}^* \triangleright \gamma \quad \Gamma \cdot y^*\colon \beta^* \vdash \mathit{form}'^* \triangleright \gamma}{\Gamma \vdash \texttt{match}\ \mathit{form}\ \texttt{(is [}\mathit{:label}\ x^*\texttt{]}\ \mathit{form}^*\texttt{)}\ \texttt{(is [}\mathit{:label}'\ y^*\texttt{]}\ \mathit{form}'^*\texttt{)} \triangleright \gamma}$$

The below rules for sequences of forms are justified by the fact that we only remember the value of the last form.

$$(\textsc{Form-empty}) \qquad \Gamma \vdash \quad \triangleright \varepsilon$$

$$(\textsc{Form-seq}) \qquad \frac{\Gamma \vdash \mathit{form}^* \triangleright \alpha^* \qquad \Gamma \vdash \mathit{form} \triangleright \beta}{\Gamma \vdash \mathit{form}^*\mathit{form} \triangleright \beta}$$

The type of an initialization form *var* := *form* is $\mathrm{Var}(\alpha)$, where α is the type of *form*.

$$(\textsc{Init-form}) \qquad \frac{\Gamma \vdash \mathit{form} \triangleright \alpha}{\Gamma \vdash \mathit{var} \texttt{ := } \mathit{form} \triangleright \mathrm{Var}(\alpha)}$$

Finally, the type of an object is a record whose components correspond to the types of the various methods. The reasoning is similar to the case of match. Notice that we also discharge assumptions on the local variables var^*.

(OBJ)

$$\frac{\Gamma \vdash initform^* \triangleright \gamma^* \quad \Gamma \cdot var^* : \gamma^* \cdot x^* : \alpha^* \vdash body \triangleright \delta \quad \Gamma \cdot var^* : \gamma^* \cdot y^* : \beta^* \vdash body' \triangleright \delta'}{\begin{array}{l}\Gamma \vdash \texttt{[object (state } initform^*\texttt{) (script (=> [}{:}label\ x^*\texttt{] } body\texttt{)} \\ \qquad\qquad\qquad \texttt{(=> [}{:}label'\ y^*\texttt{) } body'\texttt{))]} \triangleright [\mathit{label}\colon \alpha^*, \mathit{label}'\colon \beta^*]\end{array}}$$

The body of a script is handled by rule FORM-SEQ.

3.3 Some Simple Examples

The result of the following deduction tells us that `greeting` (Section 2.3) is an object capable of receiving a single message labelled with ***hello*** and whose only argument is a name capable of receiving *at least* a message labelled with ***hi***, without arguments.

$$\cfrac{\cfrac{\{\texttt{reply-to}\colon [\mathit{hi}\colon, \ldots]\} \vdash \texttt{reply-to} \triangleright [\mathit{hi}\colon, \ldots]}{\{\texttt{reply-to}\colon [\mathit{hi}\colon, \ldots]\} \vdash \texttt{reply-to <== :hi} \triangleright \mathrm{Nil}}}{\vdash \texttt{[[object}\cdots\texttt{]} \triangleright [\mathit{hello}\colon [\mathit{hi}\colon, \ldots]]}$$

To derive a type for the `counter` object of Section 2.3, we start by the initialization form. Let Γ be the basis $\{\texttt{0}\colon \mathrm{Int}, \texttt{succ}\colon [\mathrm{Int}\ [\mathit{val}\colon \mathrm{Int}]]\}$, for Int some type. By rules, ID and INIT-FORM we have the following deduction.

$$\frac{\Gamma \vdash \texttt{0} \triangleright \mathrm{Int}}{\Gamma \vdash \texttt{count := 0} \triangleright \mathsf{Var}(\mathrm{Int})}$$

Then we analyze the body of method increment. Notice that the now-message `succ <== count` does indeed stand for `succ <== [:val count]`, since all messages must be labelled. (We assumed that messages with no explicit label are labelled with `:val`.) Assuming Γ' to be the basis $\Gamma \cup \{\texttt{count}\colon \mathrm{Var}(\mathrm{Int})\}$, we have the following deduction by rules ID, VAR, NOW-MSG and ASSIGN.

$$\cfrac{\cfrac{\Gamma' \vdash \texttt{succ} \triangleright [\mathrm{Int}\ [\mathit{val}\colon \mathrm{Int}]] \qquad \Gamma' \vdash \texttt{count} \triangleright \mathrm{Int}}{\Gamma' \vdash \texttt{succ <== count} \triangleright \mathrm{Int}}}{\Gamma' \vdash \texttt{count := succ <== count} \triangleright \mathrm{Nil}}$$

Method `:value` is simpler. Again notice that the now-message `reply-to <== count` indeed stands for `reply-to <== [:val count]`. Let Γ'' be the basis $\Gamma \cup \{\texttt{reply-to}\colon [\mathit{val}\colon \mathrm{Int}, \ldots]\}$. The following derivation is obtained from rules ID, VAR and PAST-MSG.

$$\frac{\Gamma'' \vdash \texttt{reply-to} \triangleright [\mathit{val}\colon \mathrm{Int}] \qquad \Gamma'' \vdash \texttt{count}\colon \mathrm{Int}}{\Gamma'' \vdash \texttt{reply-to <== count}\colon \mathrm{Nil}}$$

Putting everything together by means of rule OBJ, we have

$$\Gamma \vdash \texttt{[object counter}\cdots\texttt{]} \triangleright [\textsf{\textit{increment}}\colon \varepsilon, \textsf{\textit{value}}\colon [\textsf{\textit{val}}\colon \mathrm{Int}, \ldots]]$$

which indeed reflects the usage of object `counter`: sent it either a message `:increment` with no arguments, or a message `:value` with an argument able to receive *at least* a message of type $\textsf{\textit{val}}$: Int.

Now-methods always have a type $\alpha^* \; [\textsf{\textit{val}}\colon \beta, \ldots]$, where $[\textsf{\textit{val}}\colon \beta, \ldots]$ is the type of the implicit `reply-to` name. We can introduce an abbreviation by writing $\alpha^* \to \beta$ instead of $\alpha^* \; [\textsf{\textit{val}}\colon \beta, \ldots]$ (cf. [5]). In this way, the type of `succ` becomes the more familiar Int $\to$ Int, which indeed reflects the applicative behaviour of the object (located at name) `succ`. Similarly, the type of object `counter` becomes

$$[\textsf{\textit{increment}}\colon \varepsilon, \textsf{\textit{value}}\colon \varepsilon \to \mathrm{Int}]$$

thus clearly showing that `:value` is a now-method.

3.4 Typing Other ABCL/1 Forms

It should be easy to derive typing rules for bracket-forms and bracket-patterns, from the encoding the encoding in Section 2.4 and the TyCO typing assignment rules.

Since a great amount of TyCO code is necessary to implement objects containing wait-for forms and express-messages, the proof of an admissible rule to type this kind of objects seems not simple. However, and perhaps not surprisingly, all the examples of encodings we tried in our system seem to follow rule OBJ, where the bodies of the methods appearing in the antecedent of the rule are not only those at the top-level but also those within wait-for forms. Essentially, the type of an object is the type of its executor. As such, for example, the window-controller object of Section 2.5 has the following type.

$$[\textsf{\textit{write}}\colon \textsf{Char}, \textsf{\textit{lock}}\colon [\textsf{\textit{locked}}\colon \varepsilon, \ldots], \textsf{\textit{release}}\colon [\textsf{\textit{unlocked}}\colon \varepsilon, \ldots]]$$

Such a type indeed reflects the usage of the window-controller object, but “flattens” all methods, in the sense that makes no distinction between top-level and wait-for methods. Such a distinction can be recovered by a separate static analysis. We could then decorate the above type with information on wait-for methods, for example by writing

$$[\textsf{\textit{write}}\colon \textsf{Char}, \textsf{\textit{lock}}\colon [\textsf{\textit{locked}}\colon \varepsilon, \ldots], \textsf{\textbf{wait}}\ \textsf{\textit{release}}\colon [\textsf{\textit{unlocked}}\colon \varepsilon, \ldots]]$$

Similarly, the clock object of Section 2.6 has a type

$$[\textsf{\textit{start}}\colon \varepsilon, \textsf{\textit{what-time}}\colon [\textsf{\textit{time}}\colon \textsf{Int}, \ldots]]$$

which tells nothing about the fact that *what-time* is indeed an express-method. Again we could recover information on express-methods and annotate the above type with this information, to obtain the below type.

$$[\textsf{\textit{start}}\colon \varepsilon, \textsf{\textbf{express}}\ \textsf{\textit{what-time}}\colon [\textsf{\textit{time}}\colon \textsf{Int}, \ldots]]$$

3.5 Principal Types and Type Inference

The type system proposed in this chapter does not possess a simple system of principal types (cf. [6]). Such a notion can be recovered by introducing constraints on the types a type-variable may be substituted for, in the form of Ohori's kinds [3]. Kinds are assigned to type-variables and take the form of $\langle l_1\colon \tilde{\alpha}_1, \ldots, l_n\tilde{\alpha}_n\rangle$. A type-variable with such a kind can only be substituted for record types of the form $[l_1\colon \tilde{\alpha}_1, \ldots, l_n\tilde{\alpha}_n, \ldots]$. In this way, for example, the principal type of the `counter` object is

$$[\mathit{increment}\colon \varepsilon, \mathit{value}\colon t]$$

where type-variable t is assigned the kind

$$\langle \mathit{val}\colon \mathrm{Int}\rangle.$$

Starting with the typing assignment system proposed in this chapter, a kinded typing assignment system for ABCL/1 programs can be defined along the lines of that for TyCO [6]. Then, an efficient algorithm to extract the principal type of an ABCL/1 program can be easily derived [6].

3.6 Comparison With Other Type Systems

Yasugi proposed a type system for an ABCL/1-like programming language [8]. Types are built from constants such as Int and Bool, by means of several constructors denoting tuples, objects, reply-destinations, and function application. The system also has a form of subtyping that provides in particular for a form of casting.

The type system proposed in this report is simpler than that of Yasugi. The reason is that every identifier in a program (be it a variable identifier, an object identifier, a primitive-function identifier, or a synchronization identifier) is seen as a name, and that types, built from a single labelled-record constructor, are assigned to names.

Also, it seems that Yasugi's type system is not complete: the programmer must help the type checker by supplying explicit type information at some points in programs. In addition, it is not clear whether the property that well-typed programs do not run into type errors holds in his system.

Bibliography

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[2] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[3] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *19th ACM Symposium on Principles of Programming Languages*, pages 154–165, 1992.

[4] Etsuya Shibayama. An ABCL kernel language and its semantics. Department of Information Science, Tokyo Institute of Technology, 1989.

[5] Vasco T. Vasconcelos. Typed concurrent objects. In *8th European Conference on Object-Oriented Programming*, LNCS. Springer-Verlag, July 1994.

[6] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *1st International Symposium on Object Technologies for Advanced Software*, volume 742 of *LNCS*, pages 460–474. Springer-Verlag, November 1993.

[7] David Walker. π-calculus semantics of object-oriented programming languages. In *Theoretical Aspects of Computer Software*, volume 526 of *LNCS*, pages 532–547. Springer-Verlag, 1992.

[8] Masahiro Yasugi. *A concurrent object-oriented programming language system for highly parallel data-driven computers and its applications.* PhD thesis, University of Tokyo, 1994. Technical report 94–7e.

[9] Akinori Yonezawa, editor. *ABCL, an Object-Oriented Concurrent System.* The MIT Press, 1990.