Typed Concurrent Objects[*]

Vasco T. Vasconcelos
Keio University
Department of Computer Science
3-14-1 Hiyoshi Kohoku-ku Yokohama 223
Japan
vasco@mt.cs.keio.ac.jp

Abstract

Based on a name-passing calculus and on its typing system the paper shows how to build several language constructors towards a strongly-typed object-oriented concurrent programming language. The basic calculus incorporates the notions of asynchronous labelled messages, concurrent objects composed of labelled methods, and a form of abstraction on processes allowing in particular to declare polymorphic classes. We introduce a notion of values as name-expressions, and show how to create subclasses of existing classes. A systematic translation of the derived constructors into the basic calculus provides for semantics and for typing rules for the new constructors.

Introduction

Concurrent objects constitute a convenient tool to describe concurrent and distributed computations. Types enforce a discipline in the usage of the programming language constructors that not only provides for partial-correctness, but also helps in writing clear programs. Furthermore, a type for a program often gives some indication on the correct usage of the program. Nevertheless, most object-oriented concurrent programming languages possess no flexible notion of types, leaving to runtime the detection of protocol errors.

The calculus of objects aims at capturing fundamental notions present in concurrent objects. Programs are built from names and labels by means of a few constructors, namely asynchronous labelled messages, objects composed of a collection of labelled methods, concurrent composition, and an operator enabling to create a new name and use it in some defined scope. Notions of agents and variables over agents increase

[*]To appear in *8th European Conference on Object-Oriented Programming*, LNCS. Springer-Verlag, July 1994.

the flexibility of the calculus. Agents are processes abstracted on a series of names. They provide for recursion and, through an ML-like let constructor, allow to declare an agent once and use it several times in a given process.

TyCO grows from the basic calculus of objects and its monomorphic typing system [17], by introducing recursive types [2, 14, 16], and a form of predicative polymorphism [3, 15]. Then, we incorporate datatype declarations, and values constructed from these declarations, in the style of ML [9], Miranda [13], and Haskell [6]. Further values are the application of values to values, and a form of name abstraction. Together, these constitute *name-expressions* evaluating to a name in a given context or continuation. A general form of encoding values into core-TyCO is presented, and admissible rules for the new constructors derived.

Core-TyCO possess a primitive notion of objects but not of classes. Making use of the let-constructor we show how to declare polymorphic classes and how to instantiate objects from classes. A notion of behaviour inheritance, by extending or replacing methods in classes, is also presented. It is shown how to translate classes and subclasses into core-TyCO; typing rules for these constructors are presented.

The paper can be best divided into two main parts: the presentation of core-TyCO and its typing system in Sections 1 and 2; and the development of the various derived constructors in Sections 3 to 9. Section 10 contains comparisons with related work and some concluding remarks.

1 The Calculus of Objects

Processes are built from an infinite set of *names*, an infinite set of *variables over agents*, and a set of *labels*. Names are denoted by $a, b, \ldots$, and also $v, x, y, \ldots$. Sequences of names of the form $x_1 \cdots x_n$, for $n \geq 0$, are abbreviated to $\tilde{x}$. We assume names are pairwise distinct in sequences $\tilde{x}$ and $\tilde{y}$. Agent-variables are denoted by $X, Y, \ldots$, and labels by $l, l_1, l_2, \ldots$. The set of *processes* is given by grammar

$$\begin{array}{lcl} P & ::= & a \lhd l : \tilde{v} \quad | \quad a \rhd [l_1 : A_1 \& \cdots \& l_n : A_n] \quad | \quad P, Q \quad | \quad |x|P \\ & | & X(\tilde{v}) \quad | \quad A(\tilde{v}) \quad | \quad \mathbf{let}\ X = A\ \mathbf{in}\ P \end{array}$$

where $P, Q, \ldots$ denote processes and the labels $l_1, \ldots, l_n$, for $n \geq 0$, are pairwise distinct. The set of *agents* is given by grammar

$$A \quad ::= \quad (\tilde{x})\, P \quad | \quad \mathbf{rec}\ X.A$$

where $A, A_1, A_2, \ldots$ denote agents.

Processes of the form $a \rhd M$ are called *objects* where a is the *location* (or identifier or address) of the object, and M represents a (finite, possibly empty) unordered collection of *methods* labelled by pairwise distinct labels. In a method of the form $l : (\tilde{x})\, P$, the sequence of names $\tilde{x}$ represents the formal parameters, while process P represents the body of the method.

Messages are processes $a \triangleleft l : \tilde{v}$ where a is the *target* and $l : \tilde{v}$ the *communication* of the message. Label l selects one particular method in the target object, while $\tilde{v}$ constitutes the actual contents of the message. Intuitively, the result of the interaction of a message $a \triangleleft l_i : \tilde{v}$ and an object $a \triangleright [l_1 : A_1 \& \cdots \& l_n : A_n]$ is the process P_i with names in $\tilde{x}_i$ replaced by those in $\tilde{v}$, if $A_i = (\tilde{x}_i)\, P_i$.

Concurrent composition P, Q denotes the process composed of P and Q running in parallel. Processes of the from $|x|P$ allow to create a new name x and to use it in a scope restricted to process P. We abbreviate a process $|x_1| \cdots |x_n|P$ to $|x_1 \cdots x_n|P$, or to $|\tilde{x}|P$.

Agents can be of two forms: *simply abstracted processes* $(\tilde{x})\, P$, or *recursively abstracted agents* **rec** $X.A$. If A is an agent of the form $(\tilde{x})\, P$, then $A(\tilde{v})$ denotes the process P with names in $\tilde{x}$ replaced by those in $\tilde{v}$. In this case, we call $A(\tilde{v})$ an *instance* of A. Similarly, an *applied agent-variable* $X(\tilde{v})$ behaves as $A(\tilde{v})$, if X is bound to A by a recursive agent or a let declaration. A process of the form **let** $X = A$ **in** P assigns agent A to X, and allows to use A multiple times in process P via X.

Scope restriction $|x|P$ and agents $(\tilde{x})\, P$ are the *name binding operators* in the calculus, binding the free occurrences of x and $\tilde{x}$ in the respective bodies P. The *set of free names* in a process P or agent A, notation $\text{fn}(P)$ or $\text{fn}(A)$, is defined accordingly. We assume the usual notion of multiple substitution of names $\tilde{v}$ for the free occurrences of names $\tilde{x}$ in a process P, notation $P\{\tilde{v}/\tilde{x}\}$, defined only if the lengths of $\tilde{x}$ and $\tilde{v}$ match, and the names in $\tilde{x}$ are pairwise distinct.

We also have bindings for agent-variables. Recursively defined agents and let-declarations constitute the *agent-variable binding operators*: **rec** $X.A$ binds X in agent A, and **let** $X = A$ **in** P binds X in process P. The *set of free agent-variables* in a process and in an agent is defined accordingly. Then, $P[X := A]$ denotes the result of replacing the free occurrences of agent-variable X by agent A in process P.

Structural congruence over processes simplifies the treatment of reduction. Following Milner [8], we define $\equiv$ to be the smallest congruence relation over processes generated by the following rules.

1. $P \equiv Q$ if P is α-convertible to Q
2. $P, Q \equiv Q, P$ and $(P, Q), R \equiv P, (Q, R)$
3. $M \equiv N$ if M is a permutation of N
4. $|x|P, Q \equiv |x|(P, Q)$ if $x \notin \text{fn}(Q)$
5. $((\tilde{x})\, P)(\tilde{v}) \equiv P\{\tilde{v}/\tilde{x}\}$ if the lengths of $\tilde{x}$ and $\tilde{v}$ match
6. **rec** $X.A \equiv A[X :=$ **rec** $X.A]$
7. **let** $X = A$ **in** $P \equiv P[X := A]$

Message application constitutes the basic communication mechanism of the calculus, representing the reception of a message by an object, followed by the selection of the appropriate method and the instantiation of the method's body. Let C be the communication $l_i : \tilde{v}$ of some message, and let M be the methods $[l_1 : A_1 \& \cdots \& l_n : A_n]$ of some object. The application of C to M, notation $M \bullet C$, is the process $P\{\tilde{v}/\tilde{x}_i\}$, provided that $1 \leq i \leq n$ and the lengths of $\tilde{v}$ and $\tilde{x}_i$ match.

The following definition of reduction relies on fact that every process can be systematically transformed into a structural congruent process of the form $|\tilde{x}|\tilde{P}$, where $\tilde{P}$ denotes the concurrent composition of messages, objects, and applied agent-variables. *One-step reduction*, notation $P \to Q$, is the smallest relation generated by the following rules.

$$\textsc{Struct} \quad \frac{P' \equiv P \qquad P \to Q \qquad Q \equiv Q'}{P' \to Q'}$$

$$\textsc{Comm} \quad |\tilde{x}|(a \triangleleft C, a \triangleright M, \tilde{P}) \to |\tilde{x}|(M \bullet C, \tilde{P})$$

2 Typing Assignment

Simple types are built from an infinite set of *type variables*, and the set of labels introduced in the previous section, by means of a *record* constructor. We use $t, t' \ldots$ to range over type-variables, $\alpha, \beta \ldots$ to range over types, and $\tilde{\alpha}, \tilde{\beta} \ldots$ to represent finite sequences of types. The set of *types* is given by grammar

$$\alpha \quad ::= \quad t \quad | \quad [l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n]$$

where $l_1, \ldots, l_n$ are pairwise distinct labels. Type $[l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n]$ is intended to denote some collection of names identifying objects containing n methods labelled with $l_1, \ldots, l_n$, and whose arguments of method l_i belong to types in $\tilde{\alpha}_i$.

Type assignments to names are formulas $x : \alpha$, for x a name and α a type. While we assign types to names, to processes we assign typings. *Typings*, denoted $\Gamma, \Delta, \ldots$, are sets of type assignments where no name occurs twice. If $\tilde{x} = x_1 \cdots x_n$ is a sequence of pairwise distinct names and $\tilde{\alpha} = \alpha_1 \cdots \alpha_n$ a sequence of types, we write $\tilde{x} : \tilde{\alpha}$ for the typing $\{x_1 : \alpha_1, \ldots x_n : \alpha_n\}$, and $\Gamma \cdot \tilde{x} : \tilde{\alpha}$ for $\Gamma \cup \tilde{x} : \tilde{\alpha}$, provided names in $\tilde{x}$ do not appear in Γ.

We say typings Γ and Δ are *compatible*, notation $\Gamma \asymp \Delta$, if $\alpha = \beta$ whenever $x : \alpha \in \Gamma$ and $x : \beta \in \Delta$. Compatible typings can be combined by a simple set-union operation.

Type assignments to agent-variables are formulas $X : \tilde{\alpha}$, for X an agent-variable and $\tilde{\alpha}$ a sequence of types. *Bases*, denoted $B, B', \ldots$, are sets of type assignments to agent-variables where no agent-variable occurs twice. Similarly to typings, we write $B \cdot X : \tilde{\alpha}$ for $B \cup \{X : \tilde{\alpha}\}$, provided X does not appear in B.

Typing assignments to processes are statements $P \triangleright \Gamma$, whereas *type assignment to agents* are statements $A \triangleright \tilde{\alpha}, \Gamma$. We write $B \vdash P \triangleright \Gamma$ (respectively $B \vdash A \triangleright \tilde{\alpha}, \Gamma$) if statement $P \triangleright \Gamma$ (respectively $A \triangleright \tilde{\alpha}, \Gamma$) is provable from basis B using the axioms and rules of the typing assignment system TA described below. Whenever $B \vdash P \triangleright \Gamma$, for some basis B and typing Γ, we say P is *typable* and call Γ a *well-typing* for P under basis B. The typing assignment system TA is composed by the following axioms and rules for processes

$$\textsc{Msg}\quad B \vdash a \lhd l_1 : \tilde{v} \;\rhd\; \{\tilde{v} \colon \tilde{\alpha}_1, a \colon [l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n]\} \quad (n \geq 1)$$

$$\textsc{Obj}\ \frac{\begin{array}{c}(a \colon [l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n] \asymp \Gamma_1 \asymp \cdots \asymp \Gamma_n, n \geq 0)\\ B \vdash A_1 \;\rhd\; \tilde{\alpha}_1, \Gamma_1 \qquad \cdots \qquad B \vdash A_n \;\rhd\; \tilde{\alpha}, \Gamma_n\end{array}}{B \vdash a \rhd [l_1 : A_1 \& \cdots \& l_n : A_n] \;\rhd\; \{a \colon [l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n]\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n}$$

$$\textsc{Scop}\ \frac{B \vdash P \;\rhd\; \Gamma \cdot x \colon \alpha}{B \vdash |x|P \;\rhd\; \Gamma} \qquad \textsc{Comp}\ \frac{B \vdash P \;\rhd\; \Gamma \qquad B \vdash Q \;\rhd\; \Delta}{B \vdash P, Q \;\rhd\; \Gamma \cup \Delta}\ (\Gamma \asymp \Delta)$$

$$\textsc{Let}\ \frac{B \vdash A \;\rhd\; \tilde{\alpha}, \Gamma \qquad B \cdot X \colon \tilde{\alpha} \vdash P \;\rhd\; \Delta}{B \vdash \textbf{let } X = A \textbf{ in } P \;\rhd\; \Gamma \cup \Delta}\ (\Gamma \asymp \Delta) \qquad \textsc{Abs}\ \frac{B \vdash P \;\rhd\; \Gamma \cdot \tilde{x} \colon \tilde{\alpha}}{B \vdash (\tilde{x})\, P \;\rhd\; \tilde{\alpha}, \Gamma}$$

$$\textsc{Weak}\ \frac{B \vdash P \;\rhd\; \Gamma}{B \vdash P \;\rhd\; \Gamma \cdot x \colon \alpha}$$

and the following axiom and rules for agents

$$\textsc{Var}\quad B \cdot X \colon \tilde{\alpha} \vdash X \;\rhd\; \tilde{\alpha}, \emptyset$$

$$\textsc{App}\ \frac{B \vdash U \;\rhd\; \tilde{\alpha}, \Gamma}{B \vdash U(\tilde{v}) \;\rhd\; \Gamma \cup \tilde{v} \colon \tilde{\alpha}}\ (\Gamma \asymp \tilde{v} \colon \tilde{\alpha}) \qquad \textsc{Rec}\ \frac{B \cdot X \colon \tilde{\alpha} \vdash A \;\rhd\; \tilde{\alpha}, \Gamma}{B \vdash \textbf{rec } X.A \;\rhd\; \tilde{\alpha}, \Gamma}$$

where, in the App-rule, U stands for an agent or an agent-variable.

There are many meaningful processes that cannot be typed in the simple typing assignment system. Processes representing natural numbers and lists are two examples of a more general class of processes containing *recurring name structures* that cannot be typed in TA. To type such processes we introduce a new type constructor $\mu t.\alpha$ denoting the infinite tree solution of the equation $t = \alpha$ [2, 14, 16]. In this way, the set of *recursive types* is defined by adding to the syntax of simple types a production $\mu t.\alpha$, for any type variable t and any recursive type α.

If α is a type, denote by α^* its associated infinite tree. An interpretation of recursive types as infinite trees naturally induces an equivalence relation $\approx$ on recursive types, by putting $\alpha \approx \beta$ if $\alpha^* = \beta^*$. The *recursive typing assignment system* TA$_\mu$ is obtained by adding rule $\approx$ to TA, and by replacing the Msg-axiom by the Msg$_\mu$-axiom.[1]

$$\approx\quad \frac{B \vdash P \;\rhd\; \Gamma \cdot x \colon \alpha}{B \vdash P \;\rhd\; \Gamma \cdot x \colon \beta}\ (\alpha \approx \beta)$$

$$\textsc{Msg}_\mu\quad B \vdash a \lhd l_1 : \tilde{v} \;\rhd\; a \colon \beta \cup \tilde{v} \colon \tilde{\alpha}_1 \quad (\beta \approx [l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n])$$

We introduce a further extension to the monomorphic type system that allows to abstract a sequence of types on a particular type-variable, and to apply a type to an

[1]The Msg$_\mu$-axiom allows to type messages whose target is contained in the message's communication, as in $a \lhd l : a$.

abstracted type. Types for agents now fall into two classes: sequences of simple types, and polymorphic types constructed using $\forall$. *Monomorphic types*, denoted $\tau, \tau', \ldots,$ are just sequences of (simple or recursive) types. *Polymorphic types* are given by the following grammar.

$$\sigma \quad ::= \quad \tau \quad | \quad \forall t.\sigma$$

As a consequence of the definition, universal quantifiers can only occur at the top level of types. A polymorphic type of the form $\forall t_1. \cdots .\forall t_n.\tau$ is abbreviated to $\forall t_1 \cdots t_n.\tau$, where $t_1, \ldots, t_n$ are the bound variables in the type. We assume the usual notion of substitution of a type α by the free occurrences of a type-variable t in a polymorphic type σ, denoted by $\sigma[\alpha/t]$. A type-variable occurs free in a typing Γ (respectively basis B) if it occurs free in some type in Γ (respectively B).

The *polymorphic typing assignment system* $\mathrm{TA}_{\mu\forall}$ is defined by the rules in TA_{μ} with the monomorphic type $\tilde{\alpha}$ in rules LET and VAR (but not OBJ, MSG$_\mu$, ABS, APP or REC) replaced by the polymorphic type σ, by allowing polymorphic types in bases (but not in typings), and with the addition of the following rules.

$$\forall\text{-INTRO}\ \frac{B \vdash A \ \triangleright\ \sigma, \Gamma}{B \vdash A \ \triangleright\ \forall t.\sigma, \Gamma}\ (t \text{ not free in } \Gamma \text{ or } B) \qquad \forall\text{-ELIM}\ \frac{B \vdash X \ \triangleright\ \forall t.\sigma, \Gamma}{B \vdash X \ \triangleright\ \sigma[\alpha/t], \Gamma}$$

We conclude this section with a brief overview of some important properties of the typing assignment system (see references [14, 15, 16, 17] for developments). *Subject-reduction* ensures that a typing for a process does not change as the process is reduced. As a corollary, typable processes do not encounter errors at runtime. We say a process P contains a possible *runtime error* if it may be reduced to a process Q of the form $|\tilde{u}|(a \triangleright M, a \triangleleft C, \tilde{P})$, and the message application $M \bullet C$ is not defined, or when $((\tilde{x})\, P)(\tilde{v})$ occurs in Q and the lengths of $\tilde{x}$ and $\tilde{v}$ do not match.

System TA (and hence systems TA_{μ} and $\mathrm{TA}_{\mu\forall}$) does not possess a simple notion of principal types. Nevertheless, this can be recovered by introducing constraints on the types a type-variable may be substituted for, in the form of Ohori's kinds [11], and by a small adjustment in the MSG-axiom [14, 17].

3 Datatype Declarations and Constructed Data

In order to build interesting programs based on the calculus proposed in the preceding sections, the first task we have to face is to construct data we can compute with. Taking advantage of labels, we introduce the notions of *datatype declarations* and *constructed data*, similar to those provided by functional programming languages such as ML [9], Miranda [13], and Haskell [6]. The data considered here is *ephemeral* in the sense that it only lasts one "use"; *persistent* data can be obtained by means of recursion, but for the sake of simplicity we only consider the ephemeral case. In the sequel, to make actual programs more natural, we write agent declarations as $X(\tilde{x}) = P$, and assume that scope restriction extends as far as possible to the right.

Boolean values can be defined as instances of the two agents below.

$$\mathsf{True(b)} = \mathsf{b}\triangleright[\mathit{val}\colon \mathsf{(reply\text{-}to)\ reply\text{-}to}\triangleleft \mathit{true}]$$
$$\mathsf{False(b)} = \mathsf{b}\triangleright[\mathit{val}\colon \mathsf{(reply\text{-}to)\ reply\text{-}to}\triangleleft \mathit{false}]$$

An object instance of True accepts a single message *val* and replies *true* stating "I am the truth value true," and similarly for instances of False. A boolean value is then an object capable of receiving a message *val* containing a name capable of receiving *at least* messages *true* and *false*, captured by the type below.

$$\mathsf{Bool} \overset{\text{def}}{=} [\mathit{val}\colon [\mathit{true}, \mathit{false}, \dots]]$$

Following the same reasoning, lists can be built from two constructors, Nil and Cons, both accepting a single message *val*.

$$\mathsf{Nil(l)} = \mathsf{l}\triangleright[\mathit{val}\colon \mathsf{(reply\text{-}to)\ reply\text{-}to}\triangleleft \mathit{nil}]$$
$$\mathsf{Cons(l\ head\ tail)} = \mathsf{l}\triangleright[\mathit{val}\colon \mathsf{(reply\text{-}to)\ reply\text{-}to}\triangleleft \mathit{cons}\colon \mathsf{head\ tail}]$$

Lists are objects capable of receiving a single message *val* and replying either *nil* or *cons*: head tail, for head an element of the list and tail a list, captured by the type

$$\mathsf{List} \overset{\text{def}}{=} \forall t.\mu u.[\mathit{val}\colon [\mathit{nil}, \mathit{cons}\colon tu, \dots]]$$

which we may as well write

$$\mathsf{List}(t) \overset{\text{def}}{=} [\mathit{val}\colon [\mathit{nil}, \mathit{cons}\colon t\,\mathsf{List}(t), \dots]].$$

This method of defining datatypes is so useful that we introduce a powerful abbreviation by simply writing the datatypes Bool and List as follows.

$$\mathsf{Bool} = [\mathit{true},\ \mathit{false}]$$
$$\mathsf{List(t)} = [\mathit{nil},\ \mathit{cons}\colon \mathsf{t\ List(t)}]$$

Similarly, natural numbers can be built from constructors zero and succ, through the declaration Nat = [*zero*, *succ*: Nat].

How do we build actual data out of type declarations? In general, constructed-values are built according to the grammar

$$\mathit{con} \quad ::= \quad a \quad | \quad l(\widetilde{\mathit{con}})$$

for a a name, l a label, and $\widetilde{\mathit{con}}$ a sequence of zero or more constructed-values. Constructed-values are translated into a name v in a given context (or continuation) P, through the encoding

$$\begin{aligned} [\![a]\!]_v P &= P[a/v] \\ [\![l(\widetilde{\mathit{con}})]\!]_v P &= [\![\widetilde{\mathit{con}}]\!]_{\tilde{v}} |v| (L(v\tilde{v}), P) \end{aligned}$$

where L is the agent-variable associated with label l by the encoding of some datatype declaration. Now, whenever we have an occurrence of a name in a non-binding position, we may as well have a constructed-value. In particular, we allow constructed-values to appear at the location of objects $\mathit{con} \triangleright M$, as arguments to instantiate agents

$X(\widetilde{con})$ or $A(\widetilde{con})$, and at the target and contents of messages $con\lhd l : \widetilde{con}$. Using the above encoding for constructed-values, part of the translation of processes with values into core-TyCO processes is as follows.

$$\begin{array}{rcl} \{\!|con \rhd M|\!\} & = & [\![con]\!]_v v \rhd \{\!|M|\!\} \\ \{\!|X(\widetilde{con})|\!\} & = & [\![con]\!]_{\tilde{v}} X(\tilde{v}) \\ \{\!|con\lhd l : \widetilde{con}|\!\} & = & [\![con\ \widetilde{con}]\!]_{v\tilde{v}} v \lhd l : \tilde{v} \end{array}$$

Notice that $\{\!|P|\!\}$ translates a general process P into a core-TyCO process, whereas $[\![con]\!]_v P$ translates a value *con*, a name v and a process P into a core-TyCO process. For example, a message

$$\mathsf{list\text{-}ops} \lhd \mathit{map}\colon\ \mathsf{f}\ \mathit{cons}(\mathsf{a}\ \mathit{cons}(\mathsf{b}\ \mathit{nil}))$$

is translated into the following process (recall that scope restriction extends as far as possible to the right).

$$\mathsf{|n|Nil(n),\ |c'|Cons(c'\ b\ n),\ |c|Cons(c\ a\ c'),\ list\text{-}ops} \lhd \mathit{map}\colon\ \mathsf{f\ c.}$$

The encoding into core-TyCO not only provides for the semantics of constructed-values, but also allows to derive an admissible system of typing assignment rules. Since constructed-values stand for names and we assign types to names, we should as well assign types to constructed-values. But because constructed-values may contain free names we must also type these names. Therefore, typing statements to constructed-values are formulas $con \ \rhd\ \alpha, \Gamma$, for α a type and Γ a typing. We write $B \vdash con \ \rhd\ \alpha, \Gamma$ is statement $con \ \rhd\ \alpha, \Gamma$ is derivable from basis B using the system of admissible rules below.

$$\textsc{Name}\quad B \vdash x \ \rhd\ \alpha, x\colon \alpha \qquad \textsc{Con}\quad \frac{B \vdash \widetilde{con} \ \rhd\ \tilde{\alpha}, \Gamma \quad B \vdash L \ \rhd\ [\mathit{val}\colon [l\colon \tilde{\alpha}, \ldots]]}{B \vdash l(\widetilde{con}) \ \rhd\ [\mathit{val}\colon [l\colon \tilde{\alpha}, \ldots]], \Gamma}$$

4 Case Expressions

Case expressions further simplify the writing of programs. Similarly to constructed-values, a case-expression evaluates to a name in a given context, and can be used wherever a non-binding name can.

$$\textbf{case}\ exp \lhd l : \widetilde{exp}\ \textbf{of}\ [l_1 : x_1 \Rightarrow exp_1 \& \ldots \& l_n : x_n \Rightarrow exp_n]$$

A case-expression of the above form evaluates to exp_i with x_i replaced by the names in the reply to message $u \lhd l : \tilde{u}r$, where r is a newly created name, and u and $\tilde{u}$ are the names resulting from the evaluation of expressions *exp* and $\widetilde{exp}$, respectively. Case expressions can be readily translated into core-TyCO.

$$\begin{array}{l} [\![\textbf{case}\ exp \lhd l : \widetilde{exp}\ \textbf{of}\ [l_1 : x_1 \Rightarrow exp_1 \& \ldots \& l_n : x_n \Rightarrow exp_n]]\!]_v P \quad = \\ \qquad [\![\widetilde{exp}\ exp]\!]_{\tilde{u}u} |r| u \lhd \mathit{val} : \tilde{u}r, r \rhd [l_1 : (x_1)\ [\![exp_1]\!]_v P \& \ldots \& l_n : (x_n)\ [\![exp_n]\!]_v P] \quad (r \text{ fresh}) \end{array}$$

Omitting label *val* and '$\lhd$' in message $\mathsf{l} \lhd \mathit{val}$, a message of the form

$$\mathsf{io}\lhd \mathit{print}{:}(\mathbf{case}\ \mathsf{l}\ \mathbf{of}\ [\mathit{nil}{:}\ \Rightarrow\ \mathit{true}\ \&\ \mathit{cons}{:}\ \_\ \_\ \Rightarrow\ \mathit{false}])$$

is translated into the following process.

$$\begin{array}{l} |\mathsf{r}|\ \mathsf{l}\lhd \mathit{val}{:}\mathsf{r},\ \mathsf{r}\rhd[\ \mathit{nil}{:}\ |\mathsf{t}|\ \mathsf{True(t)},\ \mathsf{io}\lhd \mathit{print}{:}\mathsf{t} \\ \qquad\qquad \&\mathit{cons}{:}\ (\_\ \_)\ |\mathsf{f}|\ \mathsf{False(f)},\ \mathsf{io}\lhd \mathit{print}{:}\mathsf{f}] \end{array}$$

Similarly to constructed values, typing assignments to case-expressions are formulas $\mathit{exp} \ \triangleright\ \alpha, \Gamma$. The typing rule below can be derived from system TA and the encoding of case-expressions.

$$\textsc{Case}\quad \frac{\begin{array}{c}(\Gamma \asymp \Delta \asymp \Gamma_1 \asymp \ldots \asymp \Gamma_n, n \geq 0) \\ B \vdash \mathit{exp}\ \triangleright\ [l{:}\ \tilde{\alpha}[l_1{:}\ \alpha_1, \ldots, l_n{:}\ \alpha_n]], \Gamma \qquad B \vdash \widetilde{\mathit{exp}}\ \triangleright\ \tilde{\alpha}, \Delta \\ B \vdash \mathit{exp}_1\ \triangleright\ \beta, \Gamma_1 \cdot x_1{:}\ \alpha_1 \qquad \cdots \qquad B \vdash \mathit{exp}_n\ \triangleright\ \beta, \Gamma_n \cdot x_n{:}\ \alpha_n\end{array}}{B \vdash \mathbf{case}\ \mathit{exp}\lhd l{:}\widetilde{\mathit{exp}}\ \mathbf{of}\ [l_1{:}x_1{\Rightarrow}\mathit{exp}_1\& \ldots]\ \triangleright\ \beta, \Gamma \cup \Delta \cup \Gamma_1 \cup \cdots \cup \Gamma_n}$$

Suppose that objects not and or reply with a *val* message when invoked with a *val* message. Assuming '$\lhd$' to be right-associative, and omitting labels *val*, we can simply write

$$\mathsf{not} \lhd \mathsf{or} \lhd \mathsf{b}\ \mathsf{b'}$$

instead of the more verbose form

$$\mathbf{case}\ \mathsf{not}\lhd \mathit{val}{:}(\mathbf{case}\ \mathsf{or}\lhd \mathit{val}{:}\ \mathsf{b}\ \mathsf{b'}\ \mathbf{of}\ [\mathit{val}{:}\ \mathsf{x} \Rightarrow \mathsf{x}])\ \mathbf{of}\ [\mathit{val}{:}\ \mathsf{y} \Rightarrow \mathsf{y}]$$

In general, we use the syntax

$$\mathit{exp}\lhd l{:}\widetilde{\mathit{exp}}$$

as an abbreviation for

$$\mathbf{case}\ \mathit{exp}\lhd l{:}\widetilde{\mathit{exp}}\ \mathbf{of}\ [\mathit{val}{:}x{\Rightarrow}x]$$

Conditional expressions are just particular instances of case-expressions. An expression of the form

$$\mathbf{if}\ \mathit{exp}\ \mathbf{then}\ \mathit{exp}_1\mathbf{else}\ \mathit{exp}_2$$

is simply an abbreviation for

$$\mathbf{case}\ \mathit{exp}\ \mathbf{of}\ [\mathit{true} \Rightarrow\ \mathit{exp}_1\&\ \mathit{false} \Rightarrow\ \mathit{exp}_2].$$

5 Functional Objects and Patterns

In this section we introduce a further derived constructor that allows to write function-like processes, postponing until the next section the last form of name-expressions. *Functional methods* are methods replying the result of the evaluation of a name-expression. The writing of such methods can be simplified to

$$l{:}\tilde{x}{\Rightarrow}l'{:}\widetilde{\mathit{exp}}$$

to be translated into the following method.

$$\{\!| l : \tilde{x} \Rightarrow l' : \widetilde{exp} |\!\} \quad = \quad l : (\tilde{x} r)\ [\![\widetilde{exp}]\!]_{\tilde{v}} r \lhd l' : \tilde{v}$$

In this way, by omitting labels *val*, processes implementing the not and the xor function, and part of an object implementing operations on lists can be coded as follows.

$$\begin{aligned}
&\mathsf{not} \rhd [\ \mathsf{x} \Rightarrow \textbf{if}\ \mathsf{x}\ \textbf{then}\ \mathit{false}\ \textbf{else}\ \mathit{true}] \\
&\mathsf{xor} \rhd [\ \mathsf{x}\ \mathsf{y} \Rightarrow \mathsf{not} \lhd \mathsf{or} \lhd \mathsf{x}\ \mathsf{y}] \\
&\mathsf{list\text{-}ops} \rhd [\ \mathit{null}\colon \mathsf{x} \Rightarrow \textbf{case}\ \mathsf{x}\ \textbf{of}\ [\mathit{nil}\colon \Rightarrow \mathit{true}\ \&\ \mathit{cons}\colon \_\ \_ \Rightarrow \mathit{false}] \\
&\qquad\qquad \&\mathit{map}\colon \ldots]
\end{aligned}$$

We can take advantage of labels and case-expressions, to define patterns like those in the functional programming languages ML, Miranda, and Haskell. For example, not not function can be defined using pattern-matching by the following expression.

$$\begin{aligned}
\mathsf{not} \rhd [\ & \mathit{true} \Longrightarrow \mathit{false} \\
\mid\ & \mathit{false} \Longrightarrow \mathit{true}]
\end{aligned}$$

The longer arrow '$\Longrightarrow$' and the separator '$|$' prevent the object from being confused with one with two methods *true* and *false*.

Functional objects as defined above can only be invoked once. In particular we cannot write recursive functions since calls in the body of a functional method result in messages whose target does not exist anymore. The problem can be easily fixed with replication. A replicated process of the form $!P$ represents as many copies of P as needed, running in parallel, captured by the structural congruence rule $!P \equiv P, !P$ [8].[2] For the encoding to work properly, constructed-values passed as arguments and used more than once in the body of the abstraction must be replicated. By using replication and patterns we can write the list map function as follows.

$$\begin{aligned}
&!\mathsf{list\text{-}ops} \rhd [\ \mathit{map}\colon \_\ \mathit{nil} \Longrightarrow \mathit{nil} \\
&\qquad\qquad\qquad \mid \mathsf{f}\ \mathit{cons}(\mathsf{h}\ \mathsf{t}) \Longrightarrow \mathit{cons}(\mathsf{f} \lhd \mathsf{h}\ \mathsf{list\text{-}ops} \lhd \mathit{map}\colon \mathsf{f}\ \mathsf{t}) \\
&\qquad\quad \& \ldots]
\end{aligned}$$

It should be possible to encode nested patterns as well. Compiling patterns into case-expressions is a well-studied subject [7] and the techniques known for functional programming languages with pattern-matching can be transposed to TyCO.

6 Name Abstraction

This section introduces the last form of name-expressions. An expression

$$\textbf{fn}\ \tilde{x} \Rightarrow l(\widetilde{exp})$$

[2]A process $!P$ can be coded as $(\textbf{rec}\ X.(\varepsilon)\ (X, P))\varepsilon$, for X fresh and ε the empty sequence of names. The encoding is known to preserve typings [15].

represents an object with a single *val*-labelled functional method. When invoked with actual arguments $\tilde{u}$, the object replies $l\!:\!\tilde{v}$, where $\tilde{v}$ is the result of evaluating $\widetilde{exp}$ with $\tilde{x}$ replaced by $\tilde{u}$. For example, if l is a list of boolean values, by omitting labels *val*, we can negate every element of the list by sending a message

$$|\mathsf{r}|\ \mathsf{list\text{-}ops}\lhd map\colon\ (\mathbf{fn}\ \mathsf{x} \Rightarrow \mathbf{if}\ \mathsf{x}\ \mathbf{then}\ \mathit{false}\ \mathbf{else}\ \mathit{true})\ \mathsf{l}\ \mathsf{r}$$

which should return a message $\mathsf{r}\lhd \mathit{val}\colon\ \mathsf{l'}$, with $\mathsf{l'}$ the location of the new list. The encoding into core-TyCO uses that of functional methods.

$$[\![\mathbf{fn}\ \tilde{x} \Rightarrow l(\widetilde{exp})]\!]_v P = |v|!v\rhd[\{\!|\mathit{val}\!:\!\tilde{x}\!\Rightarrow\! l\!:\!\widetilde{exp}|\!\}], P$$

The typing rule for name-abstraction is depicted below.

$$\textsc{Name-abs}\quad \frac{B \vdash \widetilde{exp}\ \rhd\ \tilde{\alpha}, \Gamma\cdot\tilde{x}\colon \tilde{\beta}}{B \vdash \mathbf{fn}\ \tilde{x} \Rightarrow l(\widetilde{exp})\ \rhd\ [\mathit{val}\colon \tilde{\beta}[l\colon \tilde{\alpha},\ldots]], \Gamma}$$

Notice that constructed-values can be seen as particular cases of name-abstractions, where the sequence $\tilde{x}$ of abstracted names is empty. For example, the constructed-value *cons*(h t) can be seen as a name-abstraction of the form **fn** $\Rightarrow$*cons*(h t). The resulting encodings are not exactly equal but should behave similarly; the typing rules agree.

This completes the presentation of name-expressions, composed of names, name-application (as case-expressions) and name-abstraction (and in particular constructed-values). Name-expressions constitute a new category in the syntax of processes; the associated typing rules can be incorporated in the typing system of core-TyCO through minor changes to rules Obj, $\textsc{Msg}_\mu$, and App.

To make clear the real applicative behaviour of name-expressions, abbreviate type $[\mathit{val}\colon \tilde{\alpha}[\mathit{val}\colon \beta,\ldots]]$ to $\tilde{\alpha} \to \beta$. Then the particular form of the Case-rule for expressions $exp\lhd\widetilde{exp}$ becomes

$$\frac{B \vdash exp\ \rhd\ \tilde{\alpha} \to \beta, \Gamma \qquad B \vdash \widetilde{exp}\ \rhd\ \tilde{\alpha}, \Delta}{B \vdash exp\lhd\widetilde{exp}\ \rhd\ \beta, \Gamma \cup \Delta} \qquad (\Gamma \asymp \Delta)$$

and that of the Name-abs-rule for expressions **fn** $\tilde{x}\Rightarrow exp$ becomes the following.

$$\frac{B \vdash exp\ \rhd\ \beta, \Gamma\cdot\tilde{x}\colon \tilde{\alpha}}{B \vdash \mathbf{fn}\ \tilde{x}\Rightarrow exp\ \rhd\ \tilde{\alpha} \to \beta, \Gamma}$$

Notice the similarity between the rules above and those of the simply typed λ-calculus. A rule similar to the particular case of the Case-rule appears in Pierce, Rémy, and Turner [12].

7 Branch Statements

Communication in TyCO is via asynchronous message passing. It is often the case that processes must wait for a reply before continuing execution. *Synchronous* invocation of methods is achieved by creating a new name to be sent together with the message. The sender process then “waits” at this name for the reply. So, for example, a process that synchronously invokes method *val* of an object x representing a list, and then *branch* into process P or Q according to the reply, may be written as follows.

$$|\mathsf{reply\text{-}to}|\mathsf{x}\lhd val\text{:}\mathsf{reply\text{-}to},\ \mathsf{reply\text{-}to}\rhd[nil\text{: } P\ \&\ cons\text{: } (\mathsf{h}\ \mathsf{t})Q]$$

Such a template is used so often that we may define a new constructor

$$\textbf{branch } a_1\lhd l_1 : \tilde{v}_1, \ldots, a_n\lhd l_n : \tilde{v}_n \textbf{ into } M,$$

for M a collection of methods, to be translated into the following process.

$$|r|a_1\lhd l_1 : \tilde{v}_1 r, \ldots, a_n\lhd l_n : \tilde{v}_n r, r\rhd M \qquad (r \text{ fresh})$$

This more general form is useful when we need to pool a number of servers and pick the first reply, ignoring further replies. Obviously, expressions can be used instead of names; the corresponding encoding is not difficult to derive. Notice that the encoding of case-expressions uses a branch-process. The forms are interrelated: **case** for expressions, and **branch** for processes.

8 Classes of Objects

Classes provide for abstraction on particular names in objects, and define a template from which executable objects can be instantiated. Objects are usually recursive and possess a distinguished name, self, commonly known as the object's identifier or location or even mail-address. As such we can define a class constructor of the form

$$\textbf{class } \mathsf{Classname}(\mathsf{var}_1 \cdots \mathsf{var}_n) = M$$

to stand as an abbreviation for

$$\textbf{rec } \mathsf{Classname}(\mathsf{self}\ \mathsf{var}_1 \cdots \mathsf{var}_n) = \mathsf{self}\ \rhd M.$$

To ensure the persistence of objects instances of classes, each method in M must recursively call Classname by means of a process $\mathsf{Classname}(\mathsf{self}\ \mathsf{var}'_1 \cdots \mathsf{var}'_n)$ somewhere in the code of the method. The fact that names $\mathsf{var}'_1, \ldots, \mathsf{var}'_n$ are not necessarily $\mathsf{var}_1, \ldots, \mathsf{var}_n$ provides for a disciplined form of assignment where all local variables are assigned at the same time, in a way reminiscent of the “become” operation in the actor model [4].

Methods in M need not recur with $\mathsf{Classname}(\mathsf{self}\ \mathsf{var}'_1 \cdots \mathsf{var}'_n)$. They may not recur at all (in which case the “existence” of the instance objects terminates), or they may instantiate one or more self-located objects of different classes, the only restriction (imposed by the typing system) being that all objects located at the same name are of the same type.

Since class is a derived constructor, we may deduce an admissible rule that allows to type a class without having to expand its definition. The following rule follows from rules Obj, Abs and Rec.

$$\textsc{Class}\quad \frac{B \cdot X\colon \beta\tilde{\alpha} \vdash M \;\triangleright\; \beta, \Gamma \cdot \mathsf{self}\,\tilde{x}\colon \beta\tilde{\alpha}}{B \vdash \textbf{class}\ X(\tilde{x}) = M \;\triangleright\; \beta\tilde{\alpha}, \Gamma}$$

Then we may abstract, by means of the $\forall$-INTRO-rule, the monomorphic type $\beta\tilde{\alpha}$ on some type variables $\tilde{t}$ (not free in Γ or B) to obtain a polymorphic type $\forall\tilde{t}.\beta\tilde{\alpha}$ for the class.

So, how do we instantiate classes to obtain objects? Since classes are just particular cases of agents, we have two alternatives given by the syntax of core-TyCO. If we only need a particular instance of the class, then the code

$$(\textbf{class}\ \mathsf{Classname}(\mathsf{var}_1 \cdots \mathsf{var}_n) = \mathsf{M})(\mathsf{obj}\ \mathsf{arg}_1 \cdots \mathsf{arg}_n)$$

instantiates an object located at name obj with local variables $\mathsf{arg}_1, \ldots, \mathsf{arg}_n$. If, on the other hand, more than one instance is needed, then we use

$$\textbf{let class}\ \mathsf{Classname}(\mathsf{var}_1 \cdots \mathsf{var}_n) = \mathsf{M}\ \textbf{in}\ \mathsf{P}$$

where P represents "the rest of the program." In particular, we allow P to instantiate as many copies of Classname as needed, through processes of the form $\mathsf{Classname}(\mathsf{obj}\ \mathsf{arg}_1 \cdots \mathsf{arg}_n)$. Instance objects need not have the same type as the class, but else may have different types instances of a polymorphic type for Classname. In this way, we can define a class Stack, polymorphic on the type of the elements stored in the stack, and instantiate in P objects representing stacks of integers, of boolean values or even stacks of stacks of some type. The rest of the program P may also contain subclass declarations, as we shall see in the next section.

As an example, suppose we want to define a class of stacks providing operations *push* and *pop*, and that we use list elems to store the elements in the stack. Such a class can be defined by a declaration

$$\textbf{class}\ \mathsf{Stack}(\mathsf{elems}) = M$$

where M comprises method *push* placing a new Cons cell at the head of the list,

$$\mathit{push}\colon\ (\mathsf{elem})\ \mathsf{Stack}(\mathsf{self}\ \mathit{cons}(\mathsf{elem}\ \mathsf{elems}))$$

and a method *pop* intended to retrieve the element at the head of the list. We have to decide the action to be taken when *pop* is invoked on an empty stack. Among many possibilities we can ignore the message, re-send the message to self (creating a kind of busy-waiting), or reply *empty*. The last alternative is depicted below.[3]

[3] If using ephemeral value constructors, when the stack is empty we cannot recur with Stack(self elems), since after evaluation elems does not exist as an object anymore.

```
pop: (reply-to) branch elems
                   into [ nil: reply-to◁empty, Stack(self nil)
                        &cons: (head tail) reply-to◁val:head, Stack(self tail)]
```

It is easy to prove that a type for class Stack is

$$\forall t.\ [push\colon t, pop\colon [empty, val\colon t, \ldots]] \cdot \mathsf{List}(t)$$

where t is the type of the elements of the stack, $[push\colon t, pop\colon [empty, val\colon t, \ldots]]$ is the type of name self, and List(t) is the type of name elems. In particular, this type says that a caller to *pop* must be ready to receive *at least* a message *val* with a name of type t, and a message *empty*.

9 Sub-classing

In this section we deal with the notion of sub-classing as a form of inheritance by *behaviour reuse*. Only minor changes to the code of a super-class are needed to ensure that recursive calls within the super-class are translated into calls to the subclass. Apart from these changes the code of the super-class is inherited "as it is" by the subclass.

We handle two forms of sub-classing: adding new methods and replacing existing methods. Both forms may be accompanied by the addition of new local variables. A possible syntax for sub-classing a class by extending its collection of methods is

$$\textbf{subclass}\ \mathsf{Subclass}(\tilde{x})\ \textbf{extends}\ \mathsf{Class}\ \textbf{by}\ M$$

where Subclass and Class are agent-variables. Such a declaration is in fact a derived form for

$$\textbf{class}\ \mathsf{Subclass}(\tilde{y}\tilde{x}) = N'\&M$$

whenever we have a declaration **class** Class($\tilde{y}$) = N, and where N' is obtained from N by replacing occurrences of recursive calls of the form Class($\tilde{v}$) by Subclass($\tilde{v}\tilde{x}$).

Since subclass is just a derived constructor, no new semantics is needed; the semantics of a subclass is just the semantics of the associated newly defined class. This may involve copying the whole code of the super-class into the subclass (with minor adjustments to recursive calls) but avoids dynamic lookup of methods and seems more realistic in a concurrent and distributed environment, where there may be many subclass instances executing in parallel.

So, to modify class Stack by adding a method *top*, we can simply write the following declaration.[4]

```
subclass TopStack extends Stack by
  [top: (reply-to) branch elems
                     into [ nil: reply-to◁empty, TopStack(self nil)
                          &cons: (head tail) reply-to◁val:head,
                                            TopStack(self cons(head tail))]]
```

[4]In both the *nil* and the *cons* case we have to rebuild elems, cf. footnote 3.

To describe the typing rule for subclasses we need a concatenation operation on record types. If α and β are record-types with disjoint labels, then $\alpha \& \beta$ is the record type containing all the components of α and β. The following rule is not admissible from the system of typing rules defined so far, but can be easily obtained from the Class-rule, and an induction on the length of derivations.

$$\textsc{Extends}\quad \frac{B \vdash X \ \triangleright\ \beta\tilde{\alpha} \qquad B \cdot Y\colon \beta'\tilde{\alpha}\tilde{\gamma} \vdash M \ \triangleright\ \alpha', \Gamma \cdot \mathsf{self}\,\tilde{y}\colon \beta'\tilde{\gamma}}{B \vdash \textbf{subclass}\ Y(\tilde{y})\ \textbf{extends}\ X\ \textbf{by}\ M \ \triangleright\ (\beta\&\beta')\tilde{\alpha}\tilde{\gamma}, \Gamma}$$

As noted before, the $\forall$-intro-rule may be used to abstract the type of the subclass. From basis $\{\mathsf{TopStack}\colon [top\colon [empty, val\colon t, \ldots]] \cdot \mathsf{List}(t)\}$ we can deduce that method *top* has a type $[empty, val\colon t, \ldots]$. Hence, by the Extends-rule followed by the $\forall$-intro-rule we have that TopStack is of type

$$\forall t.\ [push\colon t, pop\colon [empty, val\colon t, \ldots], top\colon [empty, val\colon t, \ldots]] \cdot \mathsf{List}(t).$$

Replying *empty* to a pop invocation on an empty stack may be an unacceptable solution, but we can't just "drop" method *pop* when the stack is empty, for danger of runtime errors. An alternative is to explicitly buffer callers to *pop* (and thus making them wait for the replies) and release these callers on *push*. Although we could get away with a single class, we present a solution with two mutually recursive classes, thus showing that objects may "become" instances of a different class.

So, we have two classes: EmptyStack and NonEmptyStack. The EmptyStack maintains a list wait of names waiting for replies to *pop*.[5] The code for *push* tests wait, eventually releasing one process with the just arrived element.

```
class EmptyStack(wait) =
  [ pop: (reply-to) EmptyStack(self cons(reply-to wait))
  &push: (elem) branch wait
                into [nil: NonEmptyStack(self cons(elem nil))
                     &cons: (head tail) head◁val:elem, EmptyStack(self tail)]]
```

Class NonEmptyStack is obtained from class Stack by replacing method *pop*. In general, the syntax to declare a subclass by updating a collection of methods is the following.

subclass Subclass($\tilde{x}$) **updates** Class **by** M

Again, this constructor is just a derived form for

class Subclass($\tilde{y}\tilde{x}$) $= N' + M$

whenever we have a declaration **class** Class($\tilde{y}$) $= N$, and where N' is obtained as in the **extends** case. "+" is an overwriting operator on collections of methods (and also on record types) such that $M + N$ contains all the methods in N plus those in M whose labels do not appear in N. In this way, NonEmptyStack may be defined as follows.

[5]A list does not implement the fairest solution but is sufficient for illustration purposes.

```
subclass NonEmptyStack updates Stack by
  [pop: (reply-to) branch elems
                   into [nil: EmptyStack(self cons(reply-to nil))
                        &cons: (head tail) reply-to◁val: head,
                                           NonEmptyStack(self tail)]]
```

The UPDATES-rule, the typing rule for the new constructor, is obtained from the EXTENDS-rule, by replacing operator "&" by "+", this time as an operator on record types. It is easy to prove that types for EmptyStack and NonEmptyStack are respectively,

$$\forall t.\ [push\colon t, pop\colon [val\colon t, \dots]] \cdot \mathsf{List}([val\colon t, \dots])$$
$$\forall t.\ [push\colon t, pop\colon [val\colon t, \dots]] \cdot \mathsf{List}(t)$$

revealing that name self has exactly the same type in both classes, and that the type of wait is indeed a list of objects capable of receiving values of type t.

10 Comparison with Related Work

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 [18]. We follow the opposite approach: given a name-passing calculus, build high-level constructors present in functional and concurrent object-oriented programming languages.

Pierce, Rémy, and Turner proposed a programming language based on the π-calculus [12]. Values in their work are built from tuples and value application. We go further in allowing general values constructed from datatype declarations, as well as a form of name-abstraction. Furthermore we present a systematic method of translating values into the basic calculus. The cited language allows to pass agents over names, a concept we did not develop.

While it is possible to encode TyCO into the π-calculus (and vice-versa) [17], labelled sums yield a much more elegant description of methods, with a direct counterpart in the type system. Both cited works [12, 18] assign 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.

Nierstrasz's proposed a name-passing calculus featuring communication of tuples and function application as primitives [10]. As demonstrated by Pierce, Rémy, and Turner, as well as the present work, function application can be cleanly incorporated in pure name-passing calculi.

Abadi and Cardelli recently proposed a basic calculus supporting builtin objects, and method invocation and override [1]. Although no form of concurrency is present, the calculus shares with TyCO the notions of builtin objects composed of labelled methods, and that of labelled method invocation. Builtin in that calculus and derived in TyCO, method override produces a modified copy of the initial object or class.

To my best knowledge there is no proposal of subclassing in the process-calculi framework. Behaviour subclassing, albeit quite simple, is in line with concurrent and distributed systems. Copying (at declaration or instantiation time) the code of the super-class into the subclass avoids dynamic lookup of methods making executable objects self contained. The typing rules for subclasses reuse the type of a super-class, avoiding having to deduce that type again.

The interplay between processes and expressions is delicate. The present work achieves a good balance by allowing any non-binding occurrence of a name in the core-calculus to be a substituted for a general name-expression. The price we pay for this duality is having, for example, case-expressions and branch-processes. Further study on the behaviour of name-expressions is called for.

Acknowledgements. I thank Prof. Tokoro for his encouragement and continuous support, and Kohei Honda for insightful comments on name-expressions. I am also grateful to a referee for suggesting a more thorough comparison with related work

References

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects. DEC-SRC. Shorter versions in European Symposium on Programming, and in Theoretical Aspects of Computer Software, 1994.

[2] Felice Cardone and Mario Coppo. Two extensions of Curry's type inference system. In *Logic and Computer Science*, pages 19–75. Academic Press limited, 1990.

[3] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[4] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal, modular actor formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[5] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *5th European Conference on Object-Oriented Programming*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.

[6] Paul Hudak, Simon Peyton Jones, Philip Wadler, et al. Report on the programming language Haskell: A non-strict, purely functional language. Version 1.2. *ACM Sigplan Notices*, 27(5), 1992.

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

[8] Robin Milner. Functions as processes. In *Automata, Language and Programming*, volume 443 of *LNCS*. Springer-Verlag, 1990. Also as Rapport de Recherche No 1154, INRIA-Sophia Antipolis, February 1990.

[9] Robin Milner and Mads Tofte. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1991.

[10] Oscar Nierstrasz. Towards an object calculus. In *Object-Based Concurrent Computing*, volume 469 of *LNCS*, pages 1–20. Springer-Verlag, 1992.

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

[12] Benjamin C. Pierce, Didier Rémy, and David N. Turner. A typed higher-order programming language based on the pi-calculus. University of Edinburgh, 1993.

[13] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *IFIP International Conference on Functional Programming and Computer Architecture*, volume 201 of *LNCS*. Springer-Verlag, 1985.

[14] Vasco T. Vasconcelos. Recursive types in a calculus of objects. CS 93–002, Keio University, November 1993. Revised version to appear in the Transactions of Information Processing Society of Japan.

[15] Vasco T. Vasconcelos. Predicative polymorphism in π-calculus. In *5th Parallel Architectures and Languages Europe*, LNCS. Springer-Verlag, July 1994.

[16] Vasco T. Vasconcelos and Kohei Honda. Principal typing-schemes in a polyadic π-calculus. In *4th International Conference on Concurrency Theory*, volume 715 of *LNCS*, pages 524–538. Springer-Verlag, August 1993. Also as Keio University Report CS 92-002.

[17] 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.

[18] 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.