a revised version of PPL2004 paper (a manuscript for JSSST journal "Computer Software"), 2004.

A Type System and Compilation Techniques for Concurrent Objects

Masahiro Yasugi
Graduate School of Informatics, Kyoto University

Abstract

Autonomous *concurrent objects* can be regarded as message receivers rather than labeled records by message senders, each having object references. The subtype relation on reference types can be determined by judging how various messages can be understood. Based on this idea, this paper presents a type system and compilation techniques for concurrent objects, implementing efficient pattern matching of first class messages, where the look-up can be performed as direct indexing of a small table, and the table size can be bounded by a number slightly more than the number of injection tags which the message value may have.

1 Introduction

In our computation/programming model[18, 17], computation is performed by a collection of autonomous, concurrently active software modules called *concurrent objects*, and the interaction between concurrent objects is performed solely via message passings. More than one concurrent object can become active simultaneously, and more than one message transmission may take place in parallel. Each concurrent object has its own single thread of control, and it may have its own memory, the contents of which can be accessed only by itself. Theoretical foundations for concurrent objects have been established by series of work in Actors[1], ABCM[9], POOL[2], and calculi of asynchronous objects[6].

A concurrent object-oriented programming language ABCL/1[18, 17] provides several types of message passing forms, including past type (namely, only request) and now type (namely, request followed by reply) message passing forms. However, it lacks some safety features (such as classes and static typing with subtyping) for avoiding runtime type errors (e.g., *message not understood*).

To fix this problem, our programming language ABCL/ST (ABCL/Statically Typed) was designed as a descendant of ABCL/1. In this language, a message is a first class datum (a value of a typed expression), and an object (or a reply destination) is the message receiver (target). Consequently, request messages and reply messages are well integrated, and types are well assigned to both past type and now type message passing forms. A message receiver is similar to a *continuation*, and the now type message passing style is similar to the *continuation passing style*. For an object to select its behavior according to a request message, the first class request message is typically a *variant* (a tagged value of disjoint union types). The injection tag should not be a *constructor* for a particular disjoint union type to realize *subtyping* for the object-oriented computation, where objects of various classes may be receivers of the same message. The duality between records and variants, and the duality between values and continuations (instead of functions) make our subtype relation somewhat simple. In addition, our type system allows *covariant overriding*, where a subtype τ_1' of a reference type τ_1 (an object type) may replace a reply message type τ_2 (corresponding to a return value type) in τ_1 with its subtype τ_2' (i.e., *covariance* with $\tau_1' \leq \tau_1$ and $\tau_2' \leq \tau_2$). Note that our type system is based on subtyping and does not employ *type variables* and *kinds*.

ABCL/ST is basically machine independent, but its performance was very important, since it was used when I worked on a concurrent object-oriented language system ABCL/EM-4[16] for a highly parallel data-driven computer EM-4[8]. Increasing static entities (e.g., classes and types) improves performance of parallel execution especially in distributed memory environments.

In this paper, we propose compilation techniques for implementing efficient pattern matching (by case expressions) of first class messages, where the look-up can be performed as direct indexing of a small table: the size of the look-up table can be bounded by a number slightly more than the number of injection tags which the message value may have. Our techniques support subtyping for object-oriented computation, but impose two restrictions on the type system, namely (1) explicit specification of the subtype relations (between disjoint-union types) with type *names* and (2) restricted coercing. The first restriction means that (multiple) inheritance of interface

types should be declared, although such *naming* can be used for a recursive type definition.

For direct indexing of small tables, a variant is implemented by a middle-level data structure including an *index* for representing an injection tag. The range of an index is a contiguous subset of non-negative integer; it is from 0 to n for a non-negative integer n. By fixing a particular disjoint-union type, an injection tag may be mapped into several indices but different injection tags are mapped into different indices; these properties are sufficient to use a look-up table for the disjoint-union type. By changing disjoint-union types, however, different injection tags may be mapped into a common index. Therefore, index adjustment is needed when disjoint-union types are changed by *direct cast* or *indirect cast*. The direct cast is directed by dataflow, including value passings, variable bindings, and variable accesses. By "indirect cast", we mean changing types of message receivers (continuations) instead of changing types of messages (values). To eliminate cast operations and wasteful decomposition/recomposition operations as much as possible, our type checking algorithm partly employs type assignment in a top-down manner on the syntax tree.

Explicit specification of the subtype relations leads a problem: that is, the restriction increases the cases in which the *least upper bound* of two types does not exist, and it may increase compile-time type errors. Our top-down type assignment is also useful to overcome this problem by assigning a maximum type.

In some reflective computing systems[15, 14], messages are reified as first class messages for realizing various communication patterns. In practice, we can employ first class messages as useful entities directly in our non-reflective language. For example, they can be used for *delegation*.

The rest of this paper is organized as follows. For preparation, we explain ABCL/ST and some examples in Section 2. Section 3 proposes our type system including types, a subtype relation and typing rules for a simplified ABCL/ST. Section 4 proposes our compilation techniques. The related work is discussed in Section 5.

2 Programming Language ABCL/ST

2.1 Class Definition

A *class definition* specifies behavior of objects of the class. In ABCL/ST, a *class definition* is written in the following form:

`[class` ⟨*class name*⟩ ⟨*interface type*⟩
`(`⟨*formal parameter list for creation*⟩`)`
`(state` ⟨*variable declaration list*⟩`)`
⟨*expression*⟩`].`

The parts of `(state ...)` may be omitted. The following is a simple example of *class definition*:

```
[class counter counter-o ((int c0))
 (state (int (c c0)))
 (script
  (=>  [:add i] [c := (+ c i)])
  (==> [:get] !c))].
```

The above program fragment defines a class named `counter`, which supports an *interface type* `counter-o` (details of *interface types* will be described later). `c0` is a formal parameter of type `int` for object creation (e.g., `(new counter 0)`) without using *constructors*, and `c` is a state variable of type `int` whose initial value is `c0`. An object of this class accepts a message `:add` to add the value of the argument to `c`, and also accepts a message `:get` to reply the current value of `c`. The form `!` returns a value to the *standard reply destination* included in `:get` message.

2.2 Message Passing

In ABCL/ST, an object could perform a message passing, if it knows the target. The target is either an object name or a reply destination. A reply destination can be generated by the following *now type* message passing.

Message passing is either *past type* or *now type*: a *past type* message passing is written in the following manner:

`[`⟨*target expression*⟩ `<=` ⟨*message expression*⟩`].`

For example, a past type message passing:

```
[obj <= 10]
```

sends a message `10` to the target that is the value of variable `obj`. The message is asynchronously sent and the sender object is not blocked.

A now type message passing is written in the following form:

`[`⟨*target expression*⟩ `<==`
`[`⟨*keyword*⟩ ⟨*actual parameter list*⟩`]].`

For example, a now type message passing to an object `obj` of `counter` class can be described as:

```
[obj <== [:get]].
```

The message is asynchronously sent and the sender object is blocked until it receives a reply. The reply value becomes the value of the message passing form itself. This message passing is similar to that of sequential OO languages, but the communication of the reply is also performed via message passing, whose target is the *standard reply destination* included in message `:get`. The *standard reply destination* is supplied to the message by the now type message passing.

The right hand side of now type message passing is not an ⟨*expression*⟩, while that of past type message passing is an ⟨*expression*⟩. This is because, in ABCL/ST, a message is always sent as a first class datum, and the *standard reply destination* is a part of the message, whereas, in ABCL/1, the *standard reply destination* is located on the outside of the message. On past type message passing, the value of ⟨*message expression*⟩ is directly sent as a message. On the other hand, on now type message passing, the form `[:get]` is transformed into `[:get` ⟨*standard reply destination*⟩`]` before its sending. The reply destination is a synchronizer for the reply message and the *continuation* of the now type message passing form.

2.3 Message Acceptance

Messages sent to an object are first stored in the message queue of the object.

The `wait-for` form:

```
(wait-for
 (=> ⟨matching pattern⟩ ⟨expression list⟩)
        ...
 (=> ⟨matching pattern⟩ ⟨expression list⟩))
```

accepts the first message that satisfies a ⟨*matching pattern*⟩ from the message queue and executes the corresponding ⟨*expression list*⟩. The value of the last ⟨*expression*⟩ of ⟨*expression list*⟩ becomes the value of the `wait-for` form. Unmatched messages remain in the message queue.

The `script` form:

```
(script
 (=> ⟨matching pattern⟩ ⟨expression list⟩)
        ...
 (=> ⟨matching pattern⟩ ⟨expression list⟩))
```

is almost the same as the `wait-for` form except that the `script` form deletes unmatched messages from its message queue and repeatedly waits for the next message after execution of an ⟨*expression list*⟩.

In the `wait-for` and `script` forms, `(==> [:get] !c)` is an abbreviation of:

```
(=> [:get std-rply-dst]
    [std-rply-dst <= c]).
```

2.4 Interface of Object

The next example tells us how to specify the *interface type*:

```
[class bias (obj real)
 ((real v) ((obj real) out))
 (script
  (=> inp [out <= (+ inp v)]))].
```

In this example, the *interface type* is `(obj real)`, which means that objects of this class accept messages of type `real`.

An object of class `bias` receives a message `inp` of type `real`, adds the bias value `v` to it, then sends the value to another object `out` of type `(obj real)`.

As we can see in this example, in ABCL/ST, messages for an object are of only a single type, and an interface type of an object is syntactically denoted by:

`(obj` ⟨*type*⟩`).`

Here, ⟨*type*⟩ is the type of messages which the object can accept.

In order to define *interface type* `counter-o` for the previous `counter` example, the following *type definition* can be used:

```
(deftype
 counter-o (obj obj-msg-counter-o)
 obj-msg-counter-o (union [:add int]
                          [:get (@ int)])).
```

In this definition, `counter-o` is defined as a type of objects whose messages are of a *disjoint-union type* `obj-msg-counter-o`, which is defined as a union of *keyword types* (a keyword type is a disjoint-union type with a single injection tag) `[:add int]` and `[:get (@ int)]`. The type of the *standard reply destination* of `:get` message is `(@ int)`, which means that the reply destination accepts a value of type `int`.

2.5 Tuples and Tagged Values

In ABCL/ST, a tuple is constructed by `[`⟨*expression*⟩$\cdots$⟨*expression*⟩`]` and a tagged value is constructed by `(`⟨*keyword*⟩ ⟨*expression*⟩`)`.

There is a syntax sugar concerning tuples and tagged values; for example, `[:tag1 10 20 30]` is an abbreviation of `(:tag1-*-*-* [10 20 30])` (# of `*` denotes the arity of the tuple).

2.6 Types

The types in ABCL/ST include:

- Basic types such as `int`, `real`, `bool`,
- Tuple types such as `[int real]`,
- Disjoint-union types, which include keyword types such as `[:tag1 int int int]` and disjoint-union names specified by type definitions (described below),
- Class types, which are specified by class definitions,
- Object types (interface types), such as `(obj [:tag1 int int int])`, which indicates that the object can accept messages of type `[:tag1 int int int]`,
- Reply destination types, such as `(@ int)`, which indicates that the reply destination can receive a reply message of type `int`.

2.7 Defining Subtype Relations

A `union` operation on disjoint-union types is used to specify a subtype relation. Keyword types, such as `[:tag1 int int int]`, are used to attach tags to values. A keyword type is a disjoint-union type with a single injection tag. The subtype relation between disjoint-union types are specified by the following type definitions[1]:

```
(deftype weekday
         (union [:mon] [:tue] [:wed]
                [:thu] [:fri]))
(deftype weekend (union [:sat] [:sun]))
(deftype week (union weekday weekend)).
```

Here, `week` is a supertype of `[:mon]`, `weekday` and `weekend`.

The subtype relation on object types whose messages are of disjoint-union types is specified by the following type definitions:

```
(deftype
  counter-o (obj obj-msg-counter-o)
  obj-msg-counter-o (union [:add int]
                           [:get (@ int)]))
(deftype
  counter-with-reset-o
      (obj obj-msg-counter-with-reset-o)
  obj-msg-counter-with-reset-o
      (union obj-msg-counter-o [:reset])).
```

[1]More mathematical treatment of these definitions is discussed in Section 3.

Here, `obj-msg-counter-with-reset-o` is a supertype of `obj-msg-counter-o`, and `counter-with-reset-o` is a subtype of `counter-o`. There are abbreviations for object type definition; for example, the above definition can be written as:

```
(deftype (obj-msg counter-o)
   (union [:add int] [:get (@ int)]))
(deftype (obj-msg counter-with-reset-o)
   (union (obj-msg counter-o) [:reset]))
```

or

```
[interface counter-o
   [:add int] [:get (@ int)]]
[interface counter-with-reset-o
   (obj-msg counter-o) [:reset]].
```

The following class definition gives a class whose interface type is a subtype of that of `counter-o`:

```
[class counter-with-reset
  counter-with-reset-o
  ((int c0))
  (state (int (c c0)))
  (script
    (=>  [:add i] [c := (+ c i)])
    (=>  [:reset] [c := c0])
    (==> [:get]   !c))].
```

The `union` operation on disjoint-union types is also used to define recursive types such as lists and trees:

```
(deftype list-of-int
  (union [:nil] [:cons int list-of-int]))
(deftype int-tree
  (union [:empty]
         [:node int-tree int int-tree])).
```

Then we can write:

```
 [x := [:cons 1 [:cons 2 [:nil]]]]
```

for a variable `x` of `list-of-int` type.

2.8 Examples

Figure 1 shows *delegation*. An object of class `C2` just forwards an unprocessed message M to an object of `C1-o` type. In addition, our type system allows a subtype `C2-o` of a reference type `C1-o` to replace a reply message type `C1-o` for reply to `:copy` message with its subtype `C2-o` (i.e., *covariance*).

```
[interface C1-o
  [:add int] [:set int]
  [:get (@ int)] [:copy (@ C1-o)]]
[interface C2-o
  (obj-msg C1-o) [:reset] [:copy (@ C2-o)]]

[class C1 C1-o ((int x))
  (script
    (=>  [:add i] [x := (+ x i)])
    (=>  [:set i] [x := i])
    (==> [:get] !x)
    (==> [:copy] !(new C1 x)))]

[class C2 C2-o ((C1-o c1))
  (script
    (=>  [:reset] [c1 <= [:set 0]])
    (==> [:copy]
         !(new C2 [c1 <== [:copy]]))
    (=>  M [c1 <= M]))]
```

Figure 1: Delegation.

3 Type System for ABCL/ST

This section presents the type system for ABCL/ST, including types, a subtype relation and typing rules for a simplified ABCL/ST.

First, in Section 3.1, for preparation, we will briefly examine two simple type systems to look at the duality between labeled records and labeled variants, and the duality between values and continuations. Then we will regard messages as values and objects as continuations and present the reason why we have chosen disjoint union types with labels to generate a subtype relation.

Next, in Section 3.2, we will point out difficulties on efficient implementation of a type system with subtyping and present two restrictions for efficiency, namely (1) explicit specification of the subtype relations with names and (2) restricted coercing. Then types and the subtype relation are presented. Finally, in Section 3.3, the typing rules are presented after defining a simplified ABCL/ST, followed by some comments for our type checking algorithm.

3.1 Preliminaries

3.1.1 Two Simple Type Systems

Subtype relations are often discussed on labeled record types. Other types with subtype relations are disjoint-union types using labels as injection tags.

Let us look at a type system $T_l(ord_N)$, which is given by the syntax:

$$\begin{array}{rcl} \tau & ::= & b \mid \neg\tau \mid \{l:\tau,\ldots,l:\tau\} \mid [l:\tau,\ldots,l:\tau] \\ b & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{real} \end{array}$$

where l ranges over a finite (N) set of label names, $Dom(ord_N)$ is the set of label names. $\{l:\tau,\ldots,l:\tau\}$ are labeled record types, whereas $[l:\tau,\ldots,l:\tau]$ are disjoint-union types with injection labels. For example, an expression of type $\{x:\texttt{real}, y:\texttt{real}\}$ denotes a record with x, y fields of type `real`, and an expression of type $[Z:\texttt{int}, R:\texttt{real}]$ denotes either an `int` value with injection label Z or a `real` value with injection label R.

Types of the form $\neg\tau$ are types for *continuations*. A τ-accepting continuation (which accepts a τ type value and executes the rest of the computation) has a type $\neg\tau$; this notation appears in [5], where $\neg\tau$ is defined as $\tau \to \bot$. Intuitively, this means that the function can be applied to the input value of type τ, but the result is never returned to the calling point.

$T_l(ord_N)$ does not have function types: $\tau_1 \to \tau_2$; instead, a continuation passing style (written as *cps*) function can be represented as a continuation which accepts a pair of the argument and a continuation for the reply, for example:

$$\neg\{\mathit{argument}:\tau_1, \mathit{reply\text{-}to}:\neg\tau_2\}.$$

$T_l(ord_N)$ does not have recursive types, but they are automatically introduced later by a naming mechanism which is required for efficient implementation.

Before giving the subtype relation, let us think about embedding of the types of $T_l(ord_N)$ into the simple type system without labels, namely $T_d(N)$, which gives us a naive implementation scheme of subtyped languages. For this embedding, a one-to-one morphism $ord_N : \{l_i\} \to \{1, 2, \ldots, N\}$ can be used.

$T_d(N)$ is given by the following syntax:

$$\begin{array}{rcl} \tau & ::= & \top \mid \bot \mid b \mid \neg\tau \mid (\tau_1 \times \tau_2 \times \cdots \times \tau_N) \mid \\ & & (\tau_1 + \tau_2 + \cdots + \tau_N) \\ b & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{real} \end{array}$$

where $(\tau_1 \times \tau_2 \times \cdots \times \tau_N)$ is an N-tuple type, and $(\tau_1 + \tau_2 + \cdots + \tau_N)$ is a disjoint sum type with N tags. The subtype relation of $T_d(N)$ is defined as:

$$\vdash_{T_d(N)} \bot \le \tau \quad \vdash_{T_d(N)} \tau \le \top \quad \vdash_{T_d(N)} b \le b$$

$$\frac{\vdash_{T_d(N)} \tau_1 \le \tau_1' \quad \ldots \quad \vdash_{T_d(N)} \tau_N \le \tau_N'}{\vdash_{T_d(N)} (\tau_1 \times \cdots \times \tau_N) \le (\tau_1' \times \cdots \times \tau_N')}$$

$$\frac{\vdash_{T_d(N)} \tau_1 \le \tau_1' \quad \ldots \quad \vdash_{T_d(N)} \tau_N \le \tau_N'}{\vdash_{T_d(N)} (\tau_1 + \cdots + \tau_N) \le (\tau_1' + \cdots + \tau_N')}$$

$$\frac{\vdash_{T_d(N)} \tau_1 \le \tau_2}{\vdash_{T_d(N)} \neg\tau_2 \le \neg\tau_1.}$$

The equality between types of $T_d(N)$ is the usual structural equality between types, and the subtype

relation satisfies:

$$\tau_1 \leq \tau_2 \wedge \tau_1 \geq \tau_2 \Longleftrightarrow \tau_1 = \tau_2.$$

Now, *embedding function* ψ_{ord_N} from types of $T_l(ord_N)$ into types of $T_d(N)$ can be defined as:

$$\begin{array}{c}
\psi_{ord_N}(b) = b \\
\psi_{ord_N}(\neg\tau) = \neg\,\psi_{ord_N}(\tau) \\
\psi_{ord_N}(\{l_1 : \tau_{11}, \ldots, l_n : \tau_{1n}\}) = (\tau_{21} \times \cdots \times \tau_{2N}) \\
\text{where } \tau_{2j} = \left\{ \begin{array}{ll} \psi_{ord_N}(\tau_{1i}) & \text{if } j = ord_N(l_i) \\ \top & \text{otherwise} \end{array} \right. \\
\psi_{ord_N}([l_1 : \tau_{11}, \ldots, l_n : \tau_{1n}]) = (\tau_{21} + \cdots + \tau_{2N}) \\
\text{where } \tau_{2j} = \left\{ \begin{array}{ll} \psi_{ord_N}(\tau_{1i}) & \text{if } j = ord_N(l_i) \\ \bot & \text{otherwise.} \end{array} \right.
\end{array}$$

The subtype relation on types of $T_l(ord_N)$ is defined by that of $T_d(N)$ in the following way:

$$\vdash_{T_l(ord_N)} \tau_1 \leq \tau_2 \stackrel{def}{\equiv} \vdash_{T_d(N)} \psi_{ord_N}(\tau_1) \leq \psi_{ord_N}(\tau_2).$$

This definition leads to the usual subtype relation.

3.1.2 Objects as Continuations

Labeled records are often used as objects, but here we will present another representation of objects. Labeled records have their dual representation of disjoint-union types together with the duality between values and continuations:

$$\begin{array}{l}
\{l_1 : \tau_1, l_2 : \tau_2, \ldots, l_n : \tau_n\} \\
\qquad \leftrightarrow \neg[l_1 : \neg\tau_1, l_2 : \neg\tau_2, \ldots, l_n : \neg\tau_n]
\end{array}$$

where a record type corresponds to a continuation which accepts one of $l_1, \ldots, l_n$ and returns a reply to the corresponding continuation of one of types $\neg\tau_1, \ldots, \neg\tau_n$.

Let us consider a record type whose l_1-field is a cps function. Then its dual representation would be:

$$\begin{array}{l}
\{l_1 : \neg\{\mathit{argument} : \tau_1, \mathit{reply\text{-}to} : \neg\tau_2\}\} \\
\qquad \leftrightarrow \neg[l_1 : \neg\neg\{\mathit{argument} : \tau_1, \mathit{reply\text{-}to} : \neg\tau_2\}].
\end{array}$$

The dual representation reveals a problem of "objects as records"; that is, there are two function-style transactions to dispatch an object's method: the caller should (1) pass l_1 and get $\neg\{\mathit{argument} : \tau_1, \mathit{reply\text{-}to} : \neg\tau_2\}$, (2) pass the argument of type τ_1 and get a reply of type τ_2. This is not problematic for sequential objects, but is for concurrent objects where we would not like to incur double of message traveling time (unless there are distributed (cached) copies). We rather prefer a single transaction:

$$\neg[l_1 : \{\mathit{argument} : \tau_1, \mathit{reply\text{-}to} : \neg\tau_2\}]$$

where l_1 is sent together with the argument.

For the reasons above, we choose continuations accepting variants for representation of objects. Regarding objects as continuations and regarding messages as values give us (1) an object-oriented view where a message is sent to its target, and (2) a uniform view on request messages and reply messages, namely both are value passing. Consequently, it is more natural using disjoint-union types for messages to incorporate the subtype relation on object types.

As we can see, a subtype relation has the dual representation both on labeled records and labeled variants. To avoid confusion with mixing these two, we substitute tuples, denoted by $(\tau, \tau, \ldots, \tau)$, for labeled records.

We should note that the type of objects proposed above is based on the external view of objects, namely, the viewpoint of the message sender to the objects. We think that the record types are more appropriate as types of objects from the internal view of objects, namely, from the viewpoint of the implementor of objects; there, the object's own properties should not be concealed from the implementor and may be manipulated for inheritance.

3.2 The Types and Subtype Relation for ABCL/ST

3.2.1 Restrictions for Efficiency

Let N be the number of all injection tags (or labels) which appear in the entire source program. Embedding $T_l(ord_N)$ into $T_d(N)$ gives us an implementation scheme of subtyped languages, where the time-complexity of a field selection or a case branch is $O(1)$ (just a direct table indexing operation), but space-complexity of $O(N)$ is required. That is, this naive implementation uses unacceptablely large tables.

Another naive implementation may use an associative list or a binary search algorithm; they would reduce the space complexity to $O(M)$ (where M be the number of injection tags which the tested value may have), but their time complexity would increase to $O(M)$ or $O(\log M)$. Hash search algorithms may improve the time complexity to $O(1)$, but its coefficient is still large.

We will propose a novel implementation where the time complexity of a look-up is $O(1)$ and its space complexity is slightly more than $O(M)$, where the look-up is performed by direct indexing. The implementation and its precise space complexity are described in Section 4.

For this complexity requirement, we use the restricted type system where the subtype relations between disjoint-union types must be given explicitly by the programmer. The system reduces overhead

by reflecting the programmer's mind via naming.

The explicit declaration of a subtype relation is performed with *union* operations on disjoint-union types. The resulting type of a union operation is a supertype of operands of the union operation:

```
(deftype weekday
   (union [:mon] [:tue] [:wed]
          [:thu] [:fri]))
(deftype weekend (union [:sat] [:sun]))
(deftype week (union weekday weekend)).
```

For example, `[:mon]` $\leq$ `weekday`, `weekday` $\leq$ `week`, and `weekend` $\leq$ `week` hold. By the definitions above, *explicit disjoint-union names* `weekday`, `weekend`, and `week` are introduced to give their names to the corresponding disjoint-union types.

The type definition operator, `union`, is not the disjoint sum. For example, someone may want `[:fri]` to be included also in `weekend`:

```
(deftype weekend
   (union [:fri] [:sat] [:sun]))
```

where `week` has 7 elements rather than 8 elements, because the `union` operation on `weekday` and `weekend` maps the two `[:fri]` elements into one.

Another restriction for the sake of high efficiency is that: for high-speed type conversion at the implementation level, the application of coercing rules, such as the one between **int** and **real**, (i.e., inferring the subtype relation without using a mapping provided by the `union` operation) is restricted as is described in the subsequent sections.

3.2.2 The Types and Subtype Relation

This section presents a type system T_r for ABCL/ST. In the type system for ABCL/ST, **obj** τ and **@** τ corresponds to $\neg\tau$ in $T_l(ord_N)$ and $T_d(N)$. The set of types (ranged over by τ) of T_r is given by the syntax:

$$
\begin{aligned}
&\tau ::= \top \mid \bot \mid b \mid c \mid u \mid (\tau, \tau, \ldots, \tau) \mid \mathbf{obj}\ \tau \mid \mathbf{@}\ \tau \\
&b ::= \mathbf{int} \mid \mathbf{real} \mid \mathbf{bool} \\
&u ::= k\ \tau \mid s \\
&F_s = \{s_1 \mapsto \{u_{11}, \ldots, u_{1m_1}\}, \ldots, \\
&\qquad\qquad s_n \mapsto \{u_{n1}, \ldots, u_{nm_n}\}\} \\
&F_c = \{c_1 \mapsto \mathbf{obj}\ \tau_1, \ldots, c_n \mapsto \mathbf{obj}\ \tau_n\}.
\end{aligned}
$$

The type equality is just the structural equality.

The keyword k ranges over injection tags. **obj** τ is a type for objects which receive messages of type τ. **@** τ is the type for reply destinations each of which receives a (single) message of type τ. c is a class name. F_s is a function which maps a disjoint-union name (s) into a set of disjoint-union types (either disjoint-union name (s) or keyword type ($k\ \tau$)). F_s is specified by the programmer. One constraint on F_s is that the following function $Kset(F_s)$ must be well-defined:

$$
\begin{aligned}
&\mathrm{Dom}(Kset(F_s)) = \{u_j\}, \\
&Kset(F_s)(s) = \bigcup_{u' \in F_s(s)} Kset(F_s)(u') \\
&Kset(F_s)(k\ \tau) = \{k\ \tau\}
\end{aligned}
$$

$Kset(F_s)$ recursively expands disjoint-union names until keyword types and finally provides the set of keyword types, which are the original inputs to union operation. For $Kset(F_s)$ to be well-defined, the recursive expansion must be terminated.

The subtype relation $\leq_u$ of T_r based on the explicit subtype relations between disjoint-union types is defined as follows:

$$
\begin{aligned}
&C \vdash \tau \leq_u \tau \qquad C \vdash u \leq_u s \ \text{ if } u \in F_s(s) \\
&\frac{C \vdash \tau_1 \leq_u \tau_1' \quad \ldots \quad C \vdash \tau_n \leq_u \tau_n'}{C \vdash (\tau_1, \ldots, \tau_n) \leq_u (\tau_1', \ldots, \tau_n')} \\
&\frac{C \vdash \tau_1 \leq_u \tau_2}{C \vdash \mathbf{obj}\ \tau_2 \leq_u \mathbf{obj}\ \tau_1} \qquad \frac{C \vdash \tau_1 \leq_u \tau_2}{C \vdash \mathbf{@}\ \tau_2 \leq_u \mathbf{@}\ \tau_1} \\
&\frac{C \vdash \tau_1 \leq_u \tau_2 \quad C \vdash \tau_2 \leq_u \tau_3}{C \vdash \tau_1 \leq_u \tau_3}
\end{aligned}
$$

where $C = (F_s, F_c)$. Of course $\leq_u$ is a partial order on types of T_r.

The subtype relation $\leq$ is defined by adding the restricted coercing rules in the following way:

$$
\begin{aligned}
&\frac{C \vdash \tau_1 \leq_u \tau_2}{C \vdash \tau_1 \leq \tau_2} \qquad \frac{C \vdash \tau_1 \leq \tau_2 \quad C \vdash \tau_2 \leq \tau_3}{C \vdash \tau_1 \leq \tau_3} \\
&\frac{C \vdash \tau_1 \leq \tau_2}{C \vdash k\ \tau_1 \leq k\ \tau_2} \\
&\frac{C \vdash \tau_1 \leq \tau_1' \quad \ldots \quad C \vdash \tau_n \leq \tau_n'}{C \vdash (\tau_1, \ldots, \tau_n) \leq (\tau_1', \ldots, \tau_n')} \\
&C \vdash \bot \leq \tau \qquad C \vdash \tau \leq \top \\
&C \vdash \mathbf{int} \leq \mathbf{real} \qquad C \vdash \mathbf{int} \leq \mathbf{bool} \\
&C \vdash \mathbf{obj}\ \tau \leq \mathbf{@}\ \tau \\
&C \vdash c \leq \mathbf{obj}\ \tau \ \text{ if } \mathbf{obj}\ \tau = F_c(c)
\end{aligned}
$$

The rule of **obj** $\tau \leq$ **@** τ implies that an object name can be used instead of a reply destination of the same message type.

These coercing rules for $\leq$ can not appear on proof tree nodes which are higher than the rules for **obj** $\tau_2 \leq_u$ **obj** τ_1, **@** $\tau_2 \leq_u$ **@** τ_1. This is mainly for efficiency; the details of the resulting efficient implementation are described in Section 4.2.

In addition to the constraint on $Kset(F_s)$, F_s is constrained to meet the following condition: for all

$s \in \mathrm{Dom}(F_s)$ and k, $\{\tau \mid k\ \tau \in \mathit{Kset}(F_s)(s)\}$ is either an empty set or a set which has a unique maximal element in terms of $\leq$. Let us define function *maxKset* as follows: $\mathit{maxKset}(u, F_s, k) = \max\{\tau \mid k\ \tau \in \mathit{Kset}(F_s)(u)\}$. Thus each s can be regarded as a disjoint union of the maximal elements with the keywords as the injection tags. Of course, these constraints would be checked by the compiler.

3.3 Typing for ABCL/ST

This section describes the typing for ABCL/ST. To simplify its description, first, we introduce a simplified ABCL/ST in Section 3.3.1. Then we describe the typing rules for the simplified ABCL/ST in Section 3.3.2.

Our type checker (described in Section 3.3.3) hardly fails on the subtype relation which forms not a lattice but a partial order without using explicit casts which can be found in C++ and Java, by using approximate minimum types and top-down type assignment.

3.3.1 A Simplified Language for Typing Rule Description

The simplified ABCL/ST for typing rule description is given by the syntax:

$$
\begin{aligned}
e ::= {} & c^{\tau} \mid x \mid (e, e, \ldots, e) \mid \#n\ e \mid k\ e \mid \textbf{out}\ e \mid \\
& \textbf{the}\ \tau\ e \mid e + e \mid e = e \mid l := e \mid \\
& e \Leftarrow e \mid \textbf{sync2}\ x\ e \mid \textbf{@}\ x \mid \\
& \textbf{let}\ \tau\ x = e\ \textbf{in}\ e \mid (e; e; \ldots; e) \mid \\
& \textbf{case}\ e\ \textbf{of}\ m \Rightarrow e\ \textbf{or} \ldots \textbf{or}\ m \Rightarrow e \mid \\
& \textbf{wait-for}\ m \Rightarrow e\ \textbf{or} \ldots \textbf{or}\ m \Rightarrow e \\
l ::= {} & x \mid (l, l, \ldots, l) \mid \#n\ l \\
m ::= {} & c^{\tau} \mid x \mid (m, m, \ldots, m) \mid k\ m \\
d ::= {} & \textbf{defclass}\ c\ \tau\ \tau\ x\ \textbf{state}\ \tau\ x = e\ \textbf{do}\ e
\end{aligned}
$$

where τ ranges over the set of types, k ranges over the set of injection tags, l ranges over the set of left expressions for assignments, m ranges over the set of pattern expressions for matching, e ranges over the set of usual expressions, and d is a class definition. $\#n\ e$ is n-th field selection of tuple expression e.

"**defclass** $c\ \tau\ \tau_0\ x_0$ **state** $\tau_1\ x_1 = e_1, \tau_2\ x_2 = e_2$ **do** e_b" defines a class named c. c is also used as a type. τ is an interface type, which is represented by **obj** τ_m. τ_m is the type of messages which objects of class c can receive. τ_m can be referred to by **mymsgtype** within the class definition. $\tau_0\ x_0$ is the parameter (and its type) for object creation. $\tau_i\ x_i = e_i (i = 1, \ldots)$ are declaration of state variables and expressions for their initial values. The created object executes e_b. The class definition updates F_c to be $F_c(c) = \tau$.

"$l := e$" denotes an assignment. "$e_t \Leftarrow e_m$" sends message e_m to the target e_t. "$e_1 = e_2$" is an equality check. Note that, in the equality check, we do not have to be worried about the lost labels via subtyping, since we employ disjoint-union types rather than record types for the subtype relation; a supertype of a disjoint-union type never lose the labels in contrast to the behavior of record types.[2]

"**sync2** x e" synchronizes a reply to x and the local evaluation of e and makes their pair as the result. This is used to implement now type message passing; x is like a block name in Common Lisp (i.e., `(block` ⟨*block name*⟩ `...` `(return-from` ⟨*block name*⟩ `1) ...)`), but instead of using "`return-from x`," "**@** x" produces the reply destination. The now type message passing form "`[x2 <== [:msg 1]]`" is implemented as "#1 **sync2** x_1 ($x_2 \Leftarrow$ *:msg* (**@** x_1, 1))", where the reply destination produced by "**@** x_1" is passed to object x_2 as the first message argument and the reply from x_2 is extracted as the first element of the resulting pair of synchronization. By producing the reply destination in the context of the request message construction, the reply type can directly be obtained for the type of the now type form.

3.3.2 The Typing Rules

The type judgment on expressions (e) is represented by 4-tuple $A, B \vdash_C e : \tau$. A is a type environment of variable identifiers, B is an environment of the synchronizing identifiers described above. τ is the assigned type. The typing rules are shown in Figure 2. The first rule in Figure 2 is the so-called *subsumption* rule, which permits an expression to have a type if the expression has its subtype.

$A_1 A_2$ and $A_1 + A_2$ which are used in the typing rule description are defined as:

$$
\begin{aligned}
& A = A_1 A_2 \iff \\
& \quad \mathrm{Dom}(A) = \mathrm{Dom}(A_1) \cup \mathrm{Dom}(A_2), \\
& \quad A(x) = \begin{cases} A_1(x) & \text{if } x \in \mathrm{Dom}(A_1) - \mathrm{Dom}(A_2) \\ A_2(x) & \text{if } x \in \mathrm{Dom}(A_2) \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
& A = A_1 + A_2 \iff \\
& \quad \mathrm{Dom}(A) = \mathrm{Dom}(A_1) \cup \mathrm{Dom}(A_2), \\
& \quad \mathrm{Dom}(A_1) \cap \mathrm{Dom}(A_2) = \emptyset, \\
& \quad A(x) = \begin{cases} A_1(x) & \text{if } x \in \mathrm{Dom}(A_1) \\ A_2(x) & \text{if } x \in \mathrm{Dom}(A_2). \end{cases}
\end{aligned}
$$

The type judgment on left hand side expressions (l) is represented by 3-tuple $A \vdash_l l : \tau$. The typing rules are in Figure 3.

[2] Despite of this good property, we still use the notion of minimum types in our type checking algorithm for the equality check expressions described in Section 3.3.3, because of the presence of coercing rules.

$$\frac{A, B \vdash_C e : \tau \quad \vdash_C \tau \leq \tau'}{A, B \vdash_C e : \tau'}$$

$$A, B \vdash_C c^\tau : \tau$$

$$A, B \vdash_C x : \tau \qquad \text{if } x \in \mathrm{Dom}(A), A(x) = \tau$$

$$\frac{A, B \vdash_C e_i : \tau_i \ (\forall i = 1, \ldots, n)}{A, B \vdash_C (e_1, e_2, \ldots, e_n) : (\tau_1, \tau_2, \ldots, \tau_n)}$$

$$\frac{A, B \vdash_C e : (\tau_1, \ldots, \tau_n, \ldots, \tau_{n+m})}{A, B \vdash_C \#n\ e : \tau_n}$$

$$\frac{A, B \vdash_C e : \tau}{A, B \vdash_C k\ e : k\ \tau}$$

$$\frac{A, B \vdash_C e : k\ \tau}{A, B \vdash_C \textbf{out}\ e : \tau}$$

$$\frac{A, B \vdash_C e : \tau}{A, B \vdash_C \textbf{the}\ \tau\ e : \tau}$$

$$\frac{A, B \vdash_C e_i : \tau_i (\forall i = 1, 2) \quad \vdash_C \tau = \text{num-lub}(\tau_1, \tau_2)}{A, B \vdash_C e_1 + e_2 : \tau}$$

$$\frac{A, B \vdash_C e_1 : \tau \quad A, B \vdash_C e_2 : \tau \quad \vdash_C \tau \neq \top}{A, B \vdash_C e_1 = e_2 : \textbf{bool}}$$

$$\frac{A \vdash_l l : \tau \quad A, B \vdash_C e : \tau}{A, B \vdash_C l := e : \tau}$$

$$\frac{A, B \vdash_C e_1 : \tau_1 \quad A, B \vdash_C e_2 : \tau_2 \quad \vdash_C \tau_2 = Msg(\tau_1)}{A, B \vdash_C e_1 \Leftarrow e_2 : \top}$$

$$\frac{A, B\{x \mapsto \{\tau'_1, \ldots, \tau'_n\}\} \vdash_C e : \tau_2 \quad \vdash_C \tau'_i \leq \tau_1 \ (\forall i)}{A, B \vdash_C \textbf{sync2}\ x\ e : (\tau_1, \tau_2)}$$

$$A, B \vdash_C @\ x : @\ \tau \qquad \text{if } x \in \mathrm{Dom}(B), \tau \in B(x)$$

$$\frac{A, B \vdash_C e_1 : \tau_1 \quad A\{x \mapsto \tau_1\}, B \vdash_C e_2 : \tau}{A, B \vdash_C \textbf{let}\ \tau_1\ x = e_1\ \textbf{in}\ e_2 : \tau}$$

$$\frac{A, B \vdash_C e_i : \tau_i \ (\forall i = 1, \ldots, n-1) \quad A, B \vdash_C e_n : \tau}{A, B \vdash_C (e_1; e_2; \ldots; e_n) : \tau}$$

$$\frac{A, B \vdash_C e_c : \tau' \quad \begin{array}{l} A_i \vdash_{C,m} m_i : \tau' \\ AA_i, B \vdash_C e_i : \tau \end{array} \ (\forall i)}{A, B \vdash_C \textbf{case}\ e_c\ \textbf{of}\ m_1 \Rightarrow e_1\ \textbf{or} \ldots \textbf{or}\ m_n \Rightarrow e_n : \tau}$$

$$\frac{A_i \vdash_{C,m} m_i : \textbf{mymsgtype} \quad AA_i, B \vdash_C e_i : \tau \ (\forall i)}{A, B \vdash_C \textbf{wait-for}\ m_1 \Rightarrow e_1\ \textbf{or} \ldots \textbf{or}\ m_n \Rightarrow e_n : \tau}$$

Figure 2: Typing Rules for Expressions

$$A \vdash_l x : \tau \ \text{ if } \tau \in \mathrm{Dom}(A), A(x) = \tau$$

$$\frac{A \vdash_l l_i : \tau_i \ (\forall i = 1, \ldots, n)}{A \vdash_l (l_1, l_2, \ldots, l_n) : (\tau_1, \tau_2, \ldots, \tau_n)}$$

$$\frac{A \vdash_l l : (\tau_1, \ldots, \tau_n, \ldots, \tau_{n+m})}{A \vdash_l \#n\ l : \tau_n}$$

Figure 3: Typing Rules for Left Hand Side Expressions

$$\frac{\vdash_C \tau \leq \tau' \quad \vdash_C \tau' \neq \top}{\{\} \vdash_{C,m} c^\tau : \tau'}$$

$$\{x \mapsto \tau\} \vdash_{C,m} x : \tau$$

$$\frac{A_i \vdash_{C,m} m_i : \tau_i \ (\forall i = 1, \ldots, n)}{A_1 + \cdots + A_n \vdash_{C,m} (m_1, \ldots, m_n) : (\tau_1, \ldots, \tau_n)}$$

$$\frac{A \vdash_{C,m} m : \tau_2 \quad \vdash_C \tau_2 = maxKset(\tau_1, F_s, k)}{A \vdash_{C,m} k\ m : \tau_1}$$

Figure 4: Typing Rules for Pattern Expressions

The type judgment on pattern expressions (m) is represented by 3-tuple $A \vdash_{C,m} m : \tau$. The typing rules are in Figure 4. In the rule for constant (the first rule in Figure 4), c^τ is compared with the value of τ's supertype. In the rule for keyword (the fourth rule in Figure 4), m is used to receive the value in the tagged value; τ_2 must be a supertype of all possible types of the carried value.

There are a lot of solutions of type assignments to a given (sub)expression which follow the typing rules described in the previous section. To perform type checking for subtyped languages, usually, a type checking algorithm, which assigns a minimum type to each (sub)expression, would be employed for sound and complete type checking. In the proposed type system, however, such a minimum type does not always exist even when a solution which satisfies the typing rules exists. This is because of the restriction of our type system which is required for efficiency; the restriction increases the cases in which the *least upper bound* of two types does not exist.

By all means, even under such a restriction, there would be a complete algorithm which computes all possible type assignments. However, ambiguity on selecting a single type from the solutions still remains. We would rather employ a type checking algorithm which eliminates such ambiguity and provides a single type for each (sub)expression as a result, because this makes it easier for the user to understand what the result of the algorithm is, and for the implementor to develop the optimizing compiler using the provided type information.

Our type checking algorithm proposed in the following is sound but not complete with respect to the typing rules in the previous section, but it is not ambiguous for selecting a single type for each (sub)expression. Furthermore, it hardly fails in spite of the restriction on the type system, because it basically assigns maximum types rather than minimum types. The minimum type approximation rules are only applied for six kinds of subexpression, namely,

functions of function applications, targets of message passings, the operands of numerical operations, the both sides of equality checks, examined parts of case expressions, and inner parts of reply-synchronization expressions. For those programs that these approximations succeed in isolating the minimum types, the present type checking algorithm is complete; and most programs written by the user seem to belong to this category. However, to further prevent the isolating processes from failing, the user can supply a proper supertype for an expression explicitly by '**the** τ e'.

The single type assignment on e is represented by 4-tuple $A, B \vdash_{C,S} e : \tau$. A is a type environment of variable identifiers, B is an environment of the synchronizing identifiers. τ is the assigned type. The single type assignment rules are shown in Figure 5.

The minimum type approximation on e is represented by 3-tuple $A \vdash_{C,M} e :: T$. A is a type environment of variable identifiers, T stands for approximate minimum type. (Such approximation is not required when the subtype relation form a lattice.) T is a set of types, and represents another set of types $UB(T) = \{\tau \mid (\forall \tau' \in T).\tau' \leq \tau\}$, where T is always normalized not to contain obviously redundant elements to represent the same set of types as $UB(T)$, s.t. $(\forall \tau \in T)(\forall \tau' \in T - \{\tau\}).\tau' \not\leq \tau$; this normalization is performed by *maximal*(). The minimum type approximation rules are shown in Figure 6. Note that some approximation rules do not see subexpressions; such subexpressions are checked later by the single type assignment rules.

3.3.3 Type Checking

To perform type checking, an elaborate unification algorithm is not necessary but both inheriting (namely, in a top-down manner on the syntax tree) of types and synthesizing (namely, in a bottom-up manner on the syntax tree) of approximate minimum types are required. The type checking algorithm is simple and almost identical to the single type assignment rules and the minimum type approximation rules. The type checking can be performed by using 4 kinds of procedures (when using side effects to assign types to (sub)expressions) on the syntax tree:

$\boldsymbol{TC}_b(A, e) = T$ synthesizes the approximate minimum type (T) according to the minimum type approximation rules.

$\boldsymbol{TC}_t(A, e, \sigma) = (B, S)$ inherits a request type (σ) and synthesizes an environment of the synchronizing identifiers (B) and a substitution (S) according to the single type assignment rules. If not fail, it assigns $S(\sigma)$ to e, where σ, S and $S(\sigma)$ are defined as follows:

$$\sigma ::= \tau \mid \textbf{size-unknown}(n, \sigma, sv) \mid \textbf{key-unknown}(\sigma, kv)$$

$$S = \{sv_1 \mapsto n_1, \ldots, sv_{n_s} \mapsto n_{n_s}, kv_1 \mapsto k_1, \ldots, kv_{n_k} \mapsto k_{n_k}\}$$

$$S(\textbf{size-unknown}(n, \sigma, sv)) = (\top, \ldots, \top, \overset{n}{\check{S}(\sigma)}\ \top, \ldots, \overset{m}{\check{\top}}) \quad \text{if } S(sv) = m$$

$$S(\textbf{key-unknown}(\sigma, kv)) = k\ S(\sigma) \quad \text{if } S(kv) = k$$

$$S(\tau) = \tau.$$

$\boldsymbol{TC}_l(l) = \tau$ synthesizes the type (τ) of a left hand expression (l) according to the left hand side typing rules. If not fail, it assigns τ to l.

$\boldsymbol{TC}_m(m, \tau) = A$ inherits a type (τ) and synthesizes a type environment of variable identifiers (A) according to typing rules for pattern expressions. If not fail, it assigns τ to m.

3.4 Examples

Our type checker accepts the following form:

```
[x := (if b [:cons 1 x] [:nil])]
```

where `x` is of `list-of-int` type (defined in Section 2.7). Our type checker first assigns the minimum type `list-of-int` to the left hand side expression (`x`) then the right hand side expression inherits the type. Type `list-of-int` is assigned to all of the `if`-expression, `[:cons 1 x]`, and `[:nil]`. then `[int list-of-int]` is assigned to `[1 x]`. In contrast, if we employ the usual minimum type assignment algorithm as in C++ and Java, the above form will cause a type error since we do not have the least upper bound of type `[:cons int list-of-int]` and type `[:nil]`.

Our type checker accepts the following now type message passing form:

```
[Obj1 <== [:get]]
```

which is an abbreviation of

```
(part 1 (sync2 x [Obj1 <= [:get (@ x)]])).
```

Our type checker first assigns the minimum type `counter-o` to `Obj1`, then the message expression inherits the type `obj-msg-counter-o`. Type `(@ int)` is assigned to `(@ x)` and which makes B satisfy $B(\texttt{x}) = \{\texttt{int}\}$. Finally, type `int` is assigned to `[Obj1 <== [:get]]`.

$$\frac{\vdash_C \tau \leq \tau'}{A, B \vdash_{C,S} c^{\tau} : \tau'}$$

$$\frac{\vdash_C \tau \leq \tau'}{A, B \vdash_{C,S} x : \tau'} \qquad \text{if } x \in \mathrm{Dom}(A), A(x) = \tau$$

$$\frac{A, B \vdash_{C,S} e_i : \tau_i \ (\forall i = 1, \ldots, n)}{A, B \vdash_{C,S} (e_1, e_2, \ldots, e_n) : (\tau_1, \tau_2, \ldots, \tau_n)}$$

$$\frac{A, B \vdash_{C,S} e : (\top, \ldots, \top, \tau_n, \top, \ldots, \top)}{A, B \vdash_{C,S} \#n\ e : \tau_n}$$

$$\frac{A, B \vdash_{C,S} e : \tau_2 \quad \vdash_C \tau_2 = \mathit{maxKset}(\tau_1, F_s, k)}{A, B \vdash_{C,S} k\ e : \tau_1}$$

$$\frac{A, B \vdash_{C,S} e : k\ \tau}{A, B \vdash_{C,S} \mathbf{out}\ e : \tau}$$

$$\frac{A, B \vdash_{C,S} e : \tau \quad \vdash_C \tau \leq \tau'}{A, B \vdash_{C,S} \mathbf{the}\ \tau\ e : \tau'}$$

$$\frac{A \vdash_{C,M} e_i :: \{\tau_i\} \quad A, B \vdash_{C,S} e_i : \tau_i \ (\forall i = 1, 2) \quad \vdash_C \tau = \text{num-lub}(\tau_1, \tau_2) \quad \vdash_C \tau \leq \tau'}{A, B \vdash_{C,S} e_1 + e_2 : \tau'}$$

$$\frac{A \vdash_{C,M} e_i :: T_i \quad A, B \vdash_{C,S} e_i : \tau \ (\forall i = 1, 2) \quad \vdash_C \{\tau\} = \mathit{maximal}(T_1 \cup T_2) \quad \vdash_C \mathbf{bool} \leq \tau'}{A, B \vdash_{C,S} e_1 = e_2 : \tau'}$$

$$\frac{A \vdash_l l : \tau \quad A, B \vdash_{C,S} e : \tau \quad \vdash_C \tau \leq \tau'}{A, B \vdash_{C,S} l := e : \tau'}$$

$$\frac{A \vdash_{C,M} e_1 :: \{\tau_1\} \quad A, B \vdash_{C,S} e_1 : \tau_1 \quad A, B \vdash_{C,S} e_2 : \tau_2 \quad \vdash_C \tau_2 = \mathit{Msg}(\tau_1)}{A, B \vdash_{C,S} e_1 \Leftarrow e_2 : \top}$$

$$\frac{A \vdash_{C,M} e :: \{\tau_2\} \quad A, B\{x \mapsto T_1\} \vdash_{C,S} e : \tau_2 \quad \vdash_C \tau \leq \tau_1 \ (\forall \tau \in T_1) \quad \vdash_C \tau_2 \leq \tau_2'}{A, B \vdash_{C,S} \mathbf{sync2}\ x\ e : (\tau_1, \tau_2')}$$

$$A, B \vdash_{C,S} @\ x : @\ \tau \qquad \text{if } x \in \mathrm{Dom}(B), \tau \in B(x)$$

$$\frac{A, B \vdash_{C,S} e_1 : \tau_1 \quad A\{x \mapsto \tau_1\}, B \vdash_{C,S} e_2 : \tau}{A, B \vdash_{C,S} \mathbf{let}\ \tau_1\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\frac{A, B \vdash_{C,S} e_i : \top \ (\forall i = 1, \ldots, n-1) \quad A, B \vdash_{C,S} e_n : \tau}{A, B \vdash_{C,S} (e_1; e_2; \ldots; e_n) : \tau}$$

$$\frac{A \vdash_{C,M} e_c :: \{\tau'\} \quad A, B \vdash_{C,S} e_c : \tau' \quad A_i \vdash_{C,m,S} m_i : \tau' \quad AA_i, B \vdash_{C,S} e_i : \tau (\forall i = 1, \ldots, n)}{A, B \vdash_{C,S} \mathbf{case}\ e_c\ \mathbf{of}\ m_1 \Rightarrow e_1\ \mathbf{or} \ldots \mathbf{or}\ m_n \Rightarrow e_n : \tau}$$

$$\frac{A_i \vdash_{C,m,S} m_i : \mathbf{mymsgtype} \quad AA_i, B \vdash_{C,S} e_i : \tau (\forall i = 1, \ldots, n)}{A, B \vdash_{C,S} \mathbf{wait\text{-}for}\ m_1 \Rightarrow e_1\ \mathbf{or} \ldots \mathbf{or}\ m_n \Rightarrow e_n : \tau}$$

Figure 5: Single Type Assignment Rules for Expressions

$$A \vdash_{C,M} c^{\tau} :: \{\tau\}$$

$$A \vdash_{C,M} x :: \{\tau\} \qquad \text{if } x \in \mathrm{Dom}(A), A(x) = \tau$$

$$\frac{A \vdash_{C,M} e_i :: T_i \ (\forall i = 1, \ldots, n)}{A \vdash_{C,M} (e_1, e_2, \ldots, e_n) :: \{(\tau_1', \tau_2', \ldots, \tau_n') \mid \tau_i' \in T_i \ (i = 1, \ldots, n)\}}$$

$$\frac{A \vdash_{C,M} e :: T \quad \vdash_C T' = \mathit{maximal}(\{\tau_n \mid (\tau_1, \ldots, \tau_n, \ldots, \tau_{n+m}) \in T\})}{A \vdash_{C,M} \#n \ e :: T'}$$

where $T = \{(\tau_{11}, \ldots, \tau_{1n}, \ldots, \tau_{1n+m}), \ldots, (\tau_{l1}, \ldots, \tau_{ln}, \ldots, \tau_{ln+m})\}$

$$\frac{A \vdash_{C,M} e :: T}{A \vdash_{C,M} k \ e :: \{k \ \tau' \mid \tau' \in T\}}$$

$$\frac{A \vdash_{C,M} e :: \{k \ \tau_1, \ldots, k \ \tau_n\}}{A \vdash_{C,M} \mathbf{out} \ e :: \{\tau_1, \ldots, \tau_n\}}$$

$$A \vdash_{C,M} \mathbf{the} \ \tau \ e :: \{\tau\}$$

$$\frac{A \vdash_{C,M} e_1 :: \{\tau_1\} \quad A \vdash_{C,M} e_2 :: \{\tau_2\} \quad \vdash_C \tau = \text{num-lub}(\tau_1, \tau_2)}{A \vdash_{C,M} e_1 + e_2 :: \{\tau\}}$$

$$A \vdash_{C,M} e_1 = e_2 :: \{\mathbf{bool}\}$$

$$\frac{A \vdash_l l : \tau}{A \vdash_{C,M} l := e :: \{\tau\}}$$

$$A \vdash_{C,M} e_1 \Leftarrow e_2 :: \{\top\}$$

$$\frac{A \vdash_{C,M} e :: \{\tau_2\} \quad A, B\{x \mapsto T'\} \vdash_{C,S} e : \tau_2 \quad \vdash_C T_1 = \mathit{maximal}(T')}{A \vdash_{C,M} \mathbf{sync2} \ x \ e :: \{(\tau_1', \tau_2) \mid \tau_1' \in T_1\}}$$

$$\frac{A\{x \mapsto \tau_1\} \vdash_{C,M} e_2 :: T}{A \vdash_{C,M} \mathbf{let} \ \tau_1 \ x = e_1 \ \mathbf{in} \ e_2 :: T}$$

$$\frac{A \vdash_{C,M} e_n :: T}{A \vdash_{C,M} (e_1; e_2; \ldots; e_n) :: T}$$

$$\frac{A \vdash_{C,M} e_c :: \{\tau'\} \quad A_i \vdash_{C,m,S} m_i : \tau' \quad AA_i \vdash_{C,M} e_i :: T_i \ (\forall i = 1, \ldots, n) \quad \vdash_C T = \mathit{maximal}(T_1 \cup \cdots \cup T_n)}{A \vdash_{C,M} \mathbf{case} \ e_c \ \mathbf{of} \ m_1 \Rightarrow e_1 \ \mathbf{or} \ldots \mathbf{or} \ m_n \Rightarrow e_n :: T}$$

$$\frac{A_i \vdash_{C,m,S} m_i : \mathbf{mymsgtype} \quad AA_i \vdash_{C,M} e_i :: T_i \ (\forall i = 1, \ldots, n) \quad \vdash_C T = \mathit{maximal}(T_1 \cup \cdots \cup T_n)}{A \vdash_{C,M} \mathbf{wait\text{-}for} \ m_1 \Rightarrow e_1 \ \mathbf{or} \ldots \mathbf{or} \ m_n \Rightarrow e_n :: T}$$

Figure 6: Minimum Type Approximation Rules for Expressions

4 Realizing Subtype Relation with Small Overhead

The high-level datum has a mapping to the corresponding middle-level data structure. In this section, we propose a mapping which realizes the subtype relation between high level types with small overhead.

We propose an implementation scheme of fast pattern matching of variants, where the time complexity of a look-up is $O(1)$ and the look-up is performed as a very fast operation: *direct indexing operation* of a table, and the space complexity for the look-up table is slightly more than $O(M)$ (Let M be the number of injection tags which may appear in the currently examined variant), but the precise number is given by function *maxidx* described later.

We assume the use of decision trees for efficient implementation of pattern matching. A tree is branched either based on injection tags or constants. In the actual implementation, injection tags are mapped into *indices* as is described later, and a tree is branched based on indices.

4.1 Representation of Variants

First, we examine the following disjoint-union types:

```
(deftype weekday
   (union [:mon] [:tue] [:wed]
          [:thu] [:fri]))
(deftype weekend (union [:sat] [:sun]))
(deftype week (union weekday weekend)).
```

The subtype relation among them can be represented as a tree in Figure 7. In the figure, each type is a subtype of its connecting upper ones; for example, `[:mon]` is a type which has only one element `[:mon]`, and a subtype of `weekday` and `week`.

Let `xw` be a variable of type `week`, `xwd` be a variable of type `weekday`, and `xwe` be a variable of type `weekend`. `xwd` may be used for *pattern matching* in the following expression:

```
(match xwd
  (=> [:mon] ...)
  (=> [:tue] ...)
  (=> [:wed] ...)
  (=> [:thu] ...)
  (=> [:fri] ...)).
```

To realize the pattern matching, the value of `xwd` must be distinguished for each keyword. We use an *index* for this purpose. The range of an index is a contiguous subset of non-negative integer; it is from 0 to n for a non-negative integer n. For `xwd`, the index ranges from 0 to 4, and '0' represents `[:mon]`, '1' represents `[:tue]` and so on. Similarly, for `xwe`, the index ranges from 0 to 1, and '0' represents `[:sat]`, '1' represents `:sun`. However, for `xw`, it would range from 0 to 6, and '5' would represent `[:sat]`, which is represented by '0' in the case of `xwe`. Thus, we can see a problem that the representation of `[:sat]` differs between `xw` and `xwe`. To solve this problem, when we perform an assignment:

```
[xw := xwe],
```

the index value must be shifted up (i.e., $0 \mapsto 5$, $1 \mapsto 6$). This can be performed by a quite simple operation which just *adds* 5 to the index value. We call this adding value (i.e., 5) *index adjustment value*.

This mechanism can be generalized: a disjoint-union type is either a disjoint-union type symbol (denoted by s) or a keyword type (denoted by $k\ \tau$). The index range of a disjoint-union type symbol s is constructed by concatenating all the ranges of its union elements (used by `union` operation) $\{u_1, u_2, \ldots, u_n\}(= F_s(s))$, where the *order* of the union elements must be consistent everywhere in the language system; the order for `union` operation in the source program specified by the programmer is used by our current compiler. On the other hand, the index range of a keyword type is just from 0 to 0. The number of elements in an index range is given by the following function which maps a disjoint-union type to an integer:

$$\begin{aligned} \mathit{maxidx}(s) &= \sum_{u' \in F_s(s)} \mathit{maxidx}(u') \\ \mathit{maxidx}(k\ \tau) &= 1. \end{aligned}$$

This function gives the size of look-up table, which is the space complexity of the look-up (pattern matching) operation.

Now, the index adjustment value, when we use a value of a disjoint union type (a source type) as a value of its supertype (a target disjoint-union type), is given by the function *index-diff* defined below, which is a multi-valued function (i.e., returns multiple integers) from a pair of source/target disjoint-union types. *index-diff* is derived by the following rules:

$$\mathit{index\text{-}diff}(u, u) = 0.$$

$$\frac{F_s(s_t) = \{u_1, \ldots, u_n\} \quad 1 \le j \le n \qquad \mathit{index\text{-}diff}(u_s, u_j) = i}{\mathit{index\text{-}diff}(u_s, s_t) = i + \sum_{k=1}^{j-1} \mathit{maxidx}(u_k)}$$

index-diff returns multiple values only when the source disjoint-union type appears more than once

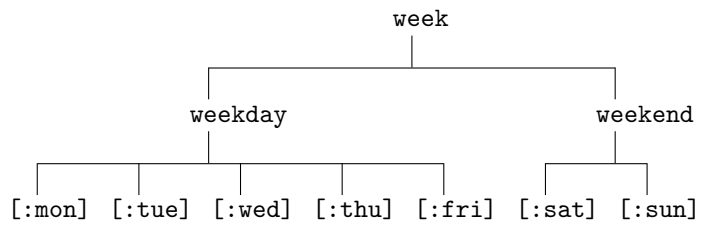

Figure 7: Subtype relation in Disjoint-Union Types.

during the decomposition of the target disjoint-union type, where decomposition means recursive application of F_s to disjoint-union type symbols started from the target disjoint-union type. Any of the returned multiple values can be used as the index adjustment value. The compiler may choose the smallest one, because some machines have quick addition instruction for small integers.

When the compiler generates a look-up table for pattern matching (by a **case** expression) of disjoint-union type data, the multiple integers returned by *index-diff* are also used to determine the entry locations on the table for each pattern. (There might be multiple table entries for a single pattern.)

So far, we have examined indices, but the value of a disjoint-union type has a carried value other than the index. We introduce *pointers* to hold the rest of the value; i.e., a disjoint-union type datum is represented by a pair ⟨*pointer, index*⟩. The indirection by pointer is required for standardizing the representation, because the size of the carried data may vary (in the above example, all the carried values are just 0-tuple, but it is not the general case).

4.2 Representation of Object References

In this section, we discuss the representation of object types. An object reference, whose type has no subtype relation, is just represented by a single object address; but an object reference, whose type has subtype relation with others, should be represented in a more elaborate way.

The subtype relation among object types is a reverse relation of their message types. For example, the subtype relation among `(obj [:mon])`, `(obj [:tue])`, `(obj [:wed])`, `(obj [:thu])`, `(obj [:fri])`, `(obj [:sat])`, `(obj [:sun])`, `(obj weekday)`, `(obj weekend)`, and `(obj week)` is shown in Figure 8.

Let `osun` be a variable of type `(obj [:sun])`, `owd` be a variable of type `(obj weekday)`, `owe` be a variable of type `(obj weekend)`, and `ow` be a variable of type `(obj week)`. The object to which `ow` is bound may perform the following pattern matching:

```
(wait-for
  (=> [:mon] ...)
  (=> [:tue] ...)
  (=> [:wed] ...)
  (=> [:thu] ...)
  (=> [:fri] ...)
  (=> [:sat] ...)
  (=> [:sun] ...)).
```

Since the type of `ow` is a subtype of the type of `owe`, the following assignment is permitted:

```
[owe := ow].
```

After this assignment, the object to which `owe` is bound is equal to the one to which `ow` is bound, and the following message send is also permitted:

```
[owe <= [:sun]].
```

The index of `[:sun]` is 1, because `owe`'s type is `(obj weekend)` and its message type should be `weekend`. However, the required index value is 6, because the actual object receives the message as type `week`.

To solve this problem, the index adjustment is necessary before the message send. Generally, this conversion can be performed with a *type conversion (identity) function*; the function may be denoted by `fn x => x : weekend -> week` in ML.[3] The function accompanies the object ID and is applied to the message before an actual message send. Thus the representation of a high-level object ID can be a pair ⟨*middle-level object ID, type conversion function*⟩.

Moreover, the type of `owe` is a subtype of the type of `osun`; the following assignments and message send are permitted:

```
[owe := ow]
[osun := owe]
[osun <= [:sun]]
```

[3]To put it more precisely, the actual target type of the type conversion function may be a supertype of `week`. But we suppose that the type is `week` in this section for simplicity.

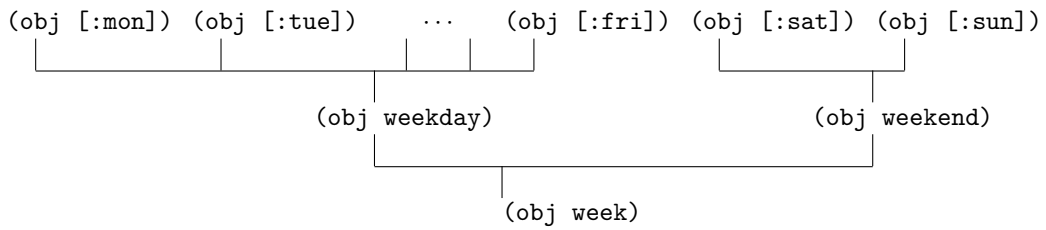

Figure 8: Subtype relation in Object Types.

In this case, the conversion should be performed by a composition function of (1)`id: [:sun] -> weekend` and (2) `id :weekend -> week`; (2) is a part of `owe`. Thus, for `[osun := owe]`, the type conversion function for `osun` should be newly composed of (1) and (2).

Let τ_s be the input type of the type conversion function and τ_t be the output type of the type conversion function. If the type conversion function is represented by a usual function, such as code block location, each message send will suffer an extra overhead of function invocation. To avoid this overhead, we would like the function to be represented by (a set of) index adjustment value(s); this is possible if $\tau_s \leq_u \tau_t$ is satisfied (not always possible if $\tau_s \leq \tau_t$). Examples are `[:sun] -> weekend`, `weekend -> week`, `[[:sun] weekend] -> [week week]`, etc. This representation also improves the speed of composition of functions significantly; for example, the composition of (1)`id: [:sun] -> weekend` and (2) `id :weekend -> week` can be performed just as an integer addition of their index adjustment values. For this reason of efficiency, as is described in Section 3.2.2, we restrict the application of coercing rules for $\leq$, such as the one between `int` and `real`.

Next, let us consider types `(obj (obj week))` and `(obj (obj weekend))`. `(obj (obj weekend))` is a subtype of `(obj (obj week))`. We need a new adjustment value to adjust index adjustment value; it may be called *"index adjustment value" adjustment value*. But we can see that all these adjustment values and index values are non-negative (small) integers.

4.3 Examples

The pattern matching in:

```
[interface counter-o
   [:add int] [:get (@ int)]]
[class counter counter-o ((int c0))
  (state (int (c c0)))
  (script
    (=>  [:add i] [c := (+ c i)])
    (==> [:get] !c))]
```

can be performed quickly, since each message for `counter-o` is represented by a pair $\langle$*pointer, index*$\rangle$ and the *index* is either 0 or 1 according to the injection tag.

Since an object of `counter-with-reset` class:

```
[interface counter-with-reset-o
   (obj-msg counter-o) [:reset]]
[class counter-with-reset
  counter-with-reset-o
  ((int c0))
  (state (int (c c0)))
  (script
    (=>  [:add i] [c := (+ c i)])
    (=>  [:reset] [c := c0])
    (==> [:get]   !c))]
```

can be used as `counter-o` type, it may receive messages for `counter-o` type. This pattern matching can also be performed quickly. The injection tag `:reset` will use 2 as *index*.

The pattern matching in:

```
[interface counter-o
   [:add int] [:get (@ int)]]
[interface counter-with-reset-o
   (obj-msg counter-o) [:reset]]
[interface counter-with-reverse-o
   (obj-msg counter-o) [:reverse]]
[interface counter-with-r-r-o
   (obj-msg counter-with-reset-o)
   (obj-msg counter-with-reverse-o)]
[class counter-with-r-r counter-with-r-r-o
  ((int c0))
  (state (int (c c0)))
  (script
    (=>  [:add i]   [c := (+ c i)])
    (=>  [:reset]   [c := c0])
    (=>  [:reverse] [c := (- c)])
    (==> [:get]     !c))]
```

can also be performed quickly, but its setting is somewhat complex. Messages for `counter-with-reset-o` will use 0, 1 and 2 as *index*

for `:add`, `:get` and `:reset`, respectively. Messages for `counter-with-reverse-o` will use 0, 1 and 2 as *index* for `:add`, `:get` and `:reverse`, respectively. So the pattern matching for `counter-with-r-r-o` uses a table of size 6 and it uses 0 and 3 for `:add` and uses 1 and 4 for `:get`. When an object reference of `counter-with-r-r-o` type is used as `counter-with-reverse-o` type, the "index adjustment value" is added by 3.

The following form can be type checked in a traditional bottom-up manner:

```
[x := [:cons 1 [:cons 2 [:nil]]]]
```

where `x` is of `list-of-int` type (defined in Section 2.7). But since both `:cons` and `:nil` will be given 0 as the index at first, it leads wasteful decomposition/recomposition operations to fix the index for `:cons` to 1 on the cast operation to type `list-of-int`. In our type system, such wasteful decomposition/recomposition operations are eliminated because our type checking algorithm basically employs type assignment in a top-down manner on the syntax tree and `:cons` is given the final index 1 from the beginning.

In Figure 1, messages for `C2-o` will use 0, 1, 2, 3, 4 and 5 as *index* for `:add`, `:set`, `:get`, `:copy`, `:reset` and `:copy`, respectively. When we see in detail, messages for `C2-o` will use 3 and 5 for `[:copy (@ C1-o)]` and `[:copy (@ C2-o)]`, respectively. Both messages are matched by the method `(=> [:copy r] [r <= (new C2 [c1 <== [:copy]])])`. Since the method expects the reply destination of type `(@ C2-o)`, a cast operation is performed on the binding of the pattern variable `r`. That is, the "index adjustment value" adjustment value is added (by 0).

5 Related Work

TyCO[12] is typed concurrent objects based on a name-passing calculus. There is also work for ABCL/1[13]. The goal of TyCO is to establish theoretical foundations for/by concurrent objects rather than to develop efficient implementation of concurrent objects.

Previous work[7, 4, 10] also uses variants for the first class messages, but a first class message in the previous work does not contain a continuation for a returned value (a reply destination in ABCL). Therefore, the previous work mainly focused on the type of a return value which is dependent on the first class message. Since a first class message in this paper contains a reply destination and replies of different types are sent to the corresponding reply destinations, we do not have to use dependent types. While *match-functions* are used in the previous work[4, 10], our approach uses *match-continuations* which returns nothing.

Polymorphic variants[3] in OCaml are another mechanism to integrate disjoint union types with polymorphism. Its type system employs structural polymorphism with *kinds*, but does not include subtyping (although it can be added). Implementation issues are also discussed in [3]. Each injection tag is mapped into a 31-bit integer and uses a look-up of $O(\log M)$ time complexity. Problematic collisions are rare since a collision is a problem only within an individual disjoint union type.

We do not think explicit specification of the subtype relations (between disjoint-union types) via *naming* is a serious problem; the naming restriction helps human understanding of programs. In C++ and Java, the same restriction exists via (multiple) *inheritance* for named classes or interfaces. Actually, C++ employs similar compilation techniques to ours for the subtype relation introduced by multiple inheritance[11]. Since our type system may assign a maximum type, overloading in C++ and Java cannot be handled in our system.

6 Conclusion

This paper presents a type system and compilation techniques for a concurrent object-oriented language ABCL/ST. In this language, a message is a first class datum (a value of a typed expression), and an object is the message target.

The type system incorporates explicit specification of subtype relations between disjoint-union types, enabling an efficient implementation scheme of look-up table. The look-up tables are used for pattern matching of data with injection tags. The size of the table can be bounded by a number slightly more than the number of injection tags which the tested value may have, and the look-up can be performed as direct indexing of the table.

Acknowledgments

I would like to thank the anonymous reviewers for their comments on improving the paper. I also thank Professor Akinori Yonezawa for his supervision and advice on this interesting research area.

References

[1] G. Agha. *Actors : A Model of Concurrent Computation in Distributed Systems.* The MIT

Press, 1987.

[2] P. America and J. Rutten. A layered semantics for a parallel object-oriented languages. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proc. of REX/FOOL, Noordwijkerhout, The Netherlands*, volume 489 of *Lecture Notes in Computer Science*, pages 91–123. Springer-Verlag, May/June 1990.

[3] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, Sept. 1998.

[4] J. Garrigue. Simple type inference for structural polymorphism. In *the 9th Workshop on Foundations of Object-Oriented Languages FOOL9*, Jan. 2002.

[5] T. G. Griffin. A formulae-as-types notion of control. In *Proceedings of 17th ACM Symposium on Principles of Programming Languages, Sanfrancisco*, pages 47–57, January 1990.

[6] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proc. of ECOOP'91, Geneva, Switzerland*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, July 1991.

[7] S. Nishimura. Static typing for dynamic messages. In *Proceedings of the 25^{th} ACM Symposium on Principles of Programming Languages*, pages 266–278, New York, 1998. ACM Press.

[8] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. In *Proc. of the 16th Annual International Symposium on Computer Architecture*, pages 46–53, June 1989.

[9] E. Shibayama. *An Object-Based Approach to Modeling Concurrent Systems*. PhD thesis, Department of Information Science, The University of Tokyo, 1991.

[10] P. Shroff and S. Smith. Type inference for first-class messages with match-functions. In *the 11th Workshop on Foundations of Object-Oriented Languages FOOL11*, Jan. 2002.

[11] B. Stroustrup. Multiple inheritance for C++. In *Proceedings of the European Unix Users Group Conference'87*, pages 189–207, May 1987.

[12] V. T. Vasconcelos. Typed concurrent objects. In *8th Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, July 1994.

[13] V. T. Vasconcelos. An operational semantics and a type for ABCL/1 based on a calculus of objects. In *Object-Oriented Computing III*, Lecture Notes, Lake Biwa, Japan, 1995. Kindai Kagaku Sha.

[14] K. Wakita. First class messages as first class continuations. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742 of *Lecture Notes in Computer Science*, pages 442–459. Springer-Verlag, 1993.

[15] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 306–315, 1988.

[16] M. Yasugi. A concurrent object-oriented programming language system for highly parallel data-driven computers and its applications. Technical Report 94-7e, Department of Information Science, Faculty of Science, University of Tokyo, Apr. 1994. (Doctoral Thesis, Mar. 1994).

[17] A. Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System — Theory, Language, Programming, Implementation and Application*. The MIT Press, 1990.

[18] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proc. of ACM Conference on OOPSLA*, pages 258–268, 1986.