A Calculus for Concurrent Objects

Paolo Di Blasio
Dipartimento di Informatica e Sistemistica
Università "La Sapienza"
e-mail: diblasio@dis.uniroma1.it

Kathleen Fisher
Computer Science Department
Stanford University
e-mail: kfisher@cs.stanford.edu

Abstract

This paper presents an imperative and concurrent extension of the functional object-oriented calculus described in [FHM94]. It belongs to the family of so-called prototype-based object-oriented languages, in which objects are created from existing ones via the inheritance primitives of object extension and method override. Concurrency is introduced through the identification of objects and processes. To our knowledge, the resulting calculus is the first concurrent object calculus to be studied. We define an operational semantics for the calculus via a transition relation between configurations, which represent snapshots of the run-time system. Our static analysis includes a type inference system, which statically detects *message-not-understood* errors, and an effect system, which guarantees that synchronization code, specified via guards, is side-effect free. We present a subject reduction theorem, modified to account for imperative and concurrent features, and type and effect soundness theorems.

1 Introduction

In the past few years, the desire to bring the benefits of object-oriented programming (modularity, re-usability and incremental design) to multiprocessor environments has led to a significant interest in concurrent object-oriented programming. The fact that objects seem to provide a suitable abstraction for concurrent programming has further encouraged research in this area. Various languages have been designed, some from scratch (*e.g.* POOL [Ame89] and ABCL [Yon90]), and others by adding concurrent features to existing object-oriented languages (*e.g.* concurrent C++ [CGH89] and Eiffel[Car93]). Despite this broad interest, the most effective combination of the object-oriented and concurrent paradigms has not yet emerged.

Considerable effort has been spent in developing theoretical frameworks for studying this issue. To date, there have been two main approaches to such studies: actor languages and process algebras. The actor model [Agh86] can easily represent concurrent objects [Agh90] and has been used as a foundation for designing various concurrent object-oriented languages (*e.g.* ABCL [Yon90] and [FA93]). Theoretical research using this model has focused on semantics and equational theories (*e.g.* [AMS92, Tal96]). The process algebra approaches to modeling concurrent object-oriented systems [PT94, Vas94, Nie92, KY94, Jon93] are often extensions of the π-calculus, obtained by adding objects and functional constructs. Researchers adopting this approach have paid particular attention to typing issues (*c.f.* [Vas94, KY94]).

Both the actor and the process algebra approaches suffer from not making the notion of an object a central concept. Because of this lack, some object-oriented features can be quite difficult to represent in these frameworks. In contrast, functional object-oriented calculi [AC94, FHM94] already have primitives for object-oriented features and so are a convenient starting point for studying concurrent object-oriented systems. Despite the naturalness of adding concurrency to such calculi, this approach has not been extensively investigated. To our knowledge, Cardelli's object-based language for distributed computation Obliq [Car95] is the only existing language that adopts this philosophy. Except for its sequential imperative core [AC95], however, no formal study of Obliq's semantics or its types has been carried out.

The general goal of this research is to establish theoretical foundations for concurrent object-oriented programming, focusing on the development of intuitive semantics and sound type systems. As a first step towards this goal, we present an imperative and concurrent extension of the calculus in [FHM94]. We believe that one of the contributions of this work is to introduce a new direction for the design of formal systems for studying concurrent object-oriented programming, that of adding concurrency primitives to object calculi.

The rest of the paper is organized as follows. In Section 2, we describe our extended language, focusing on the design choices we made in defining our calculus. In Section 3 we present an example program-fragment to illustrate our language. Section 4 describes an operational semantics for the language, and Section 5 presents its static type system. Section 6 contains the technical results of the paper, including a subject reduction theorem, modified to account for imperative and concurrent features. Type soundness follows as a corollary. Section 7 concludes with some notes on future work.

2 The language: an overview

In this section, we give an overview of our concurrent object-calculus, focusing on the design choices we made in its development.

Typing. The usefulness of static type systems in increasing the reliability and readability of programs, in detecting compile-time type errors, and in providing useful compile-time information is widely recognized. Hence, we chose to define a static type system for our calculus. Reassuringly, only straightforward modifications to the type system of the functional object calculi of [FHM94] were required. One such modification was the incorporation of an effect system [LG88] to insure that the guards we use to define synchronization code are side-effect free (See below). The potential to leverage prior work in this fashion is one of the appeals of designing concurrent object-oriented calculi by extending sequential functional ones.

Prototype-based calculus. The object model of our calculus uses a prototype-based approach to represent inheritance. In other words, new objects are created from existing ones via the inheritance primitives of object extension and method override. A consequence of this approach is that the methods of each object are embedded in the object itself, which is important for managing method lookup when objects are physically distributed on a network. In our syntax the expression $\langle\rangle$ creates an empty object, $\langle ob \longleftarrow\!\!+\; m = method\rangle$ extends object *ob* with new method m whose code is specified in *method*, and $\langle ob \leftarrow m = method\rangle$ replaces *ob*'s m-method with new code *method*. We adopt these inheritance primitives instead of the more complex operators defined in Obliq (*e.g. clone*) because they can be easily encoded.

Processes as objects. Objects represent a suitable and appealing abstraction for the notion of a process, so we introduce concurrency by identifying objects and processes. It is hoped that this unification might simplify the process of writing concurrent code. More definitely, it allows us to use our object primitives to create and activate processes, thus reducing the number of primitives in the calculus.

Communication mechanisms. Concurrent object-oriented languages that identify the notions of object and process use three kind of communication: synchronous, asynchronous and eager invocation [WKH92]. In our calculus we directly support synchronous and asynchronous method invocation because both are interesting and neither is encodable in the other without adding more syntax. We do not directly provide eager invocation because it can be derived via an encoding of future variables as objects. In our syntax, expression $ob \Leftarrow_a m(arg)$ sends message m asynchronously with argument arg to object *ob*. The corresponding synchronous invocation is $ob \Leftarrow_s m(arg)$.

Synchronization constraints. In a concurrent object-oriented language, at any given time an object may only be able to respond to a subset of its entire set of messages without losing its internal integrity. For example, a buffer object cannot meaningfully respond to a *put* message if its internal storage is full. Instead, it must wait until a *get* message causes some of its space to become free. Such restrictions on the availability of methods are called *synchronization constraints* because they affect the order in which methods are executed. Code that controls method availability is called *synchronization code* [MY93]. We use guards for this purpose, because they provide one of the most natural ways to define synchronization code and they require minimal additional syntax. (See [Yon90] for another use of guards.) In our language, each method consists of a guard and a method body, $method = when(g)\ body$. To insure that our guards behave properly, our static analysis guarantees that guards return boolean values and cause no side effects.

Protection. A crucial point in the calculus is the distinction between *self* and *non-self-inflicted* operations. According to the definitions introduced in [Car95], method overriding $\langle ob \leftarrow m = method\rangle$ and invocation $ob \Leftarrow m(arg)$[1] are *self-inflicted* iff *ob* is the same object as the *self*-parameter of the current method. Otherwise the operation is *non-self-inflicted*[2]. Method overriding has different semantics in these two cases. In the self-inflicted case, $\langle ob \leftarrow m = method\rangle$ replaces the method m of the object *ob*. This operation is the only way to update the state of an object in our calculus. The non-self-inflicted overriding has a

[1] The symbol $\Leftarrow$ indicates either synchronous and asynchronous method invocation.

[2] Note that self-inflicted method extension is prevented by the type system.

cloning semantics, meaning that we first create a copy of the object ob and then replace the m method with the new $method$[3]. There are two reasons for these choices. First, we want to provide a form of protection against external writing operations. By insuring that only self-inflicted operations can modify their host object, we can safeguard internal invariants of objects. Secondly, the cloning semantics allows us to support *depth inheritance* (via method overriding) and *width inheritance* (via object extension).

Serialized objects. As we will formalize in Section 4, an object can be in either an idle or a busy state. Non-self-inflicted operations can be executed only if their target object is in the idle state. This restriction gives each object a serialized structure: at any time a given object is involved in at most one thread of computation. This single-threadedness helps maintain object invariants, since we are guaranteed that once an object starts a computation, it will not be interrupted by an outside request until it has finished its computation. Self-inflicted operations do not have to wait for their target object to become idle. The methods of an object need to be able to access other host object methods without immediately causing deadlock. This freedom maintains the single-threadedness of objects, since self-inflicted operations continue in the same thread.

As can be seen from the preceding paragraphs, the notion of self-infliction is an important concept. Unfortunately, whether an operation is self-inflicted or not can generally only be determined at run-time. Currently, our type system does not approximate this distinction. We leave this question to future work.

3 Example

To provide some intuition for this calculus, we give example producer, consumer and one-slot buffer objects. As we will see formally in the next section, the language extends the untyped λ-calculus with object primitives. Here as a notational convenience, we adopt some syntactic conventions. We write $\langle m_1 = when(e_1)e'_1, \ldots, m_k = when(e_k)e'_k\rangle$ for $\langle \ldots \langle\langle\rangle \longleftrightarrow m_1 = when(e_1)e'_1\rangle \ldots \longleftrightarrow m_k = when(e_k)e'_k\rangle$. We omit the guard in a method definition if it is the constant $true$, *i.e.*, if the method is always available. If we have a method invocation with no parameters, we write $e_1 \Leftarrow m$ instead of $e_1 \Leftarrow m(nil)$, where nil is a constant with type $unit$. We introduce the semicolon operator $e_1; e_2$ as syntactic sugar for $((\lambda z.\, \lambda x.\, x)e_1)e_2$, which has the expected meaning for sequencing as our semantics reflects call-by-value evaluation. Finally we use integer constants and their related operations.

The following code creates an empty one-slot buffer object:

$$
\begin{array}{llll}
buffer = & \langle & x & = \lambda self.\, \lambda arg.\, 0, \\
 & & size & = \lambda self.\, \lambda arg.\, 0, \\
 & & put & = when(\lambda self.(self \Leftarrow_s size) = 0) \\
 & & & \quad \lambda self.\, \lambda v.\langle\langle self \leftarrow x = \lambda s.\, \lambda a.\, v\rangle \leftarrow size = \lambda s.\, \lambda a.\, 1\rangle, \\
 & & get & = when(\lambda self.(self \Leftarrow_s size) = 1) \\
 & & & \quad \lambda self.\, \lambda arg.\langle self \leftarrow size = \lambda s.\, \lambda a.\, 0\rangle; self \Leftarrow_s x \quad \rangle
\end{array}
$$

Both guards and method bodies occasionally need to access other methods of their host

[3]Non-self-inflicted method extension has the same semantics.

object. To grant this access, we write them as functions from a $self$ parameter to their actual code. The operational semantics binds these parameters to the host object when the corresponding methods are invoked. The x method represents the storage of the buffer. The $size$ method indicates whether or not the buffer is full by storing either a 0 (for empty) or a 1 (for full). Both of these methods have constantly $true$ guards, which we omit for clarity. The *put* method is only available if the buffer is currently empty, which its guard checks by comparing its current size to 0. When available, *put* stores a new value into the buffer's storage x and sets the $size$ to 1. Finally, the *get* method, which is only available if the buffer is full, sets the current $size$ to 0 and returns the value contained in the storage x.

When an object is created, it is associated with an address generated by the system. Future interactions with that object occur via the generated address, which may be thought of as the name of the object. For expository purposes, we will assume that the name generated for the above *buffer* object is *buff*.

We may write a producer object that interacts with the *buff* object as follows:

$$producer \;=\; \langle m \;=\; \lambda self.\, \lambda arg.((\lambda y.\, \mathit{buff} \Leftarrow_a put(y))(value\_production())) \,;\, self \Leftarrow_a m\rangle$$

Invoking a producer object's m method causes a non-terminating computation to start. During each iteration of this computation, the producer gets a new value by calling the function $value\_production$ and then stores this value in the buffer by asynchronously sending *buff* the message *put*. We will assume the name generated for the *producer* object is *prod*.

Our consumer object has the same structure:

$$consumer \;=\; \langle n \;=\; \lambda self.\, \lambda arg.((\lambda y.\, consume\_value(y))(\mathit{buff} \Leftarrow_s get)) \,;\, self \Leftarrow_a n\rangle$$

Once its n method is invoked, the consumer starts a non-terminating computation. During each iteration, the consumer gets a new value from the buffer by synchronously sending the *get* message to *buff*. It then consumes the returned value by calling the function $consume\_value$. We will assume the address generated for this *consumer* object is *cons*.

We illustrate the computational behavior of our objects using the following simplified evaluation rule that reflects the operational semantics defined precisely below:

$$\langle m_1 = when(e'_1)e_1, \ldots, m_k = when(e'_k)e_k\rangle \Leftarrow m_i \;\longrightarrow\; e_i\, \langle m_1 = when(e'_1)e_1, \ldots, m_k = when(e'_k)e_k\rangle$$

This rule allows us to evaluate a message send by retrieving the appropriate method body from the object and applying it to the entire object itself.

Using the rule above and call-by-value $beta$-reduction, we may evaluate the message send $prod \Leftarrow_a m$, assuming $value\_production() \longrightarrow v_1$:

$$\begin{array}{lll}
prod \Leftarrow_a m & \longrightarrow & (\lambda self.\, \lambda arg.((\lambda y.\, \mathit{buff} \Leftarrow_a put(y))(value_prodction())) \,;\, self \Leftarrow_a m)\, prod\, nil \\
& \longrightarrow & ((\lambda y.\, \mathit{buff} \Leftarrow_a put(y))(value_production())) \,;\, prod \Leftarrow_a m) \\
& \longrightarrow & (\mathit{buff} \Leftarrow_a put(v_1)) \,;\, prod \Leftarrow_a m \\
& \longrightarrow & prod \Leftarrow_a m
\end{array}$$

The asynchronous method invocation $\mathit{buff} \Leftarrow_a put(v_1)$ (which is a non-self-inflicted operation) is put in the queue of pending messages, and the computation continues, sending message m to the object *prod*. Note that the recursive structure of objects easily supports non-terminating computation. When the buffer is free to receive the message $put(v_1)$, it evaluates the guard for the method *put*:

$$(\lambda self.(self \Leftarrow_s size) = 0)\, buff \longrightarrow (buff \Leftarrow_s size) = 0$$

If the buffer is empty, $(buff \Leftarrow_s size) = 0$ will evaluate to true. In this case, the $put(v_1)$ message is removed from the queue and the method body is executed. Otherwise, the message remains in the queue. The put method reduces as follows:

$$\begin{array}{lll} buff \Leftarrow_a put(v_1) & \longrightarrow & (\lambda self.\, \lambda v.\langle\langle self \leftarrow x = \lambda s.\, \lambda a.\, v\rangle \leftarrow size = \lambda s.\, \lambda a.\, 1\rangle)\, buff\, v_1 \\ & \longrightarrow & \langle\langle buff \leftarrow x = \lambda s.\, \lambda a.\, v_1\rangle \leftarrow size = \lambda s.\, \lambda a.\, 1\rangle \\ & \longrightarrow & \langle buff \leftarrow size = \lambda s.\, \lambda a.\, 1\rangle \\ & \longrightarrow & buff \end{array}$$

This evaluation overrides the methods x and $size$ (both self-inflicted operations), so the object $buff$ now has the following structure, modulo α-conversion:

$$\begin{array}{lllll} buff: & \langle & x & = & \lambda self.\, \lambda arg.\, v_1, \\ & & size & = & \lambda self.\, \lambda arg.\, 1, \quad put \; = \; \ldots, \quad get \; = \; \ldots \;\rangle \end{array}$$

The evaluation of the consumer method n $(cons \Leftarrow_a n)$ is similar to the previous case.

4 Operational Semantics

We formalize the operational semantics of the calculus as a transition relation on *configurations*, which can be thought of as global snapshots of the run-time system. A configuration contains the collection of all created objects and all pending messages. Formally, a *configuration* $\langle\!\langle\, \alpha \mid \mu \,\rangle\!\rangle$ consists of an *object soup* α, containing run-time objects, and a collection of *pending asynchronous messages* μ. More precisely, μ is a finite map from integers to pending messages. As a notational simplification, we use a single collection of pending messages instead of having a smaller queue for each object.

An *object* is represented at run-time as a triple $(a, \eta_a, [S_a])$, where a is the object's *address*, η_a is its *method table*, and S_a its *state*. A method table is a partial function from the set of method names M to guarded expressions of the form $when(e_2)e_3$. The state S_a can be either idle ($[I]$) or busy ($[te_a]$), in which case the expression te_a represents the remaining computation. An object passes from the idle to a busy state in response to either a synchronous or an asynchronous method invocation. At the end of the resulting computation, it returns to the idle state.

4.1 Formal Specification

In this section, we introduce the notation needed to formally describe the operational semantics of the calculus.

Language expressions:

$$\begin{array}{lll} e & ::= & x \mid c \mid a \mid \lambda x.\, e \mid e_1 e_2 \\ & & \mid \langle\rangle \mid e_1 \Leftarrow_a m(e_2) \mid e_1 \Leftarrow_s m(e_2) \\ & & \mid \langle e_1 \leftarrow\!\!+ \; m = when(e_2)e_3\rangle \mid \langle e_1 \leftarrow m = when(e_2)e_3\rangle \end{array}$$

$$te \; ::= gs(e, a) \mid ret(e, a) \mid ga(e, c) \mid nonret(e)$$

In the set of expressions e, x is a variable, c is a constant symbol, a is an object address, $\lambda x.\, e$ is a lambda abstraction, and $e_1 e_2$ is function application. The remaining sytactic forms are the object primitives described in Section 2. All expressions e except a may occur in source programs. The top-level expressions te appear as the states of busy objects in object triples. They allow us to determine if the expression we are reducing corresponds to a guard evaluation or to a method body application and further, if the reduction is in response to a synchronous or an asynchronous invocation. (We will see an example of this in the next section, where these top-level expressions are explained in more detail.)

To describe transitions internal to an object, we need to uniquely decompose each non-value expression into a reduction context filled with a redex. With this intent, we define values, redexes and reduction contexts.

Values:

$$v \ ::= \ x \mid c \mid a \mid \lambda x.\, e$$

Top Values:

$$tv \ ::= gs(v, a) \mid ret(v, a) \mid ga(v, c) \mid nonret(v)$$

Redexes:

$$\begin{aligned} e_{rdx} \ ::= \ & v_1 v_2 \mid \langle\rangle \mid v_1 \Leftarrow_a m(v_2) \mid v_1 \Leftarrow_s m(v_2) \\ & \mid \langle v_1 \longleftrightarrow m = when(e) v_2 \rangle \mid \langle v_1 \leftarrow m = when(e) v_2 \rangle \end{aligned}$$

Inner Reduction Contexts:

$$\begin{aligned} r_{in} \ ::= \ & \Box \mid r_{in} e \mid v\, r_{in} \\ & \mid r_{in} \Leftarrow_a m(e) \mid v \Leftarrow_a m(r_{in}) \mid r_{in} \Leftarrow_s m(e) \mid v \Leftarrow_s m(r_{in}) \\ & \mid \langle r_{in} \longleftrightarrow m = when(e) e_1 \rangle \mid \langle v \longleftrightarrow m = when(e) r_{in} \rangle \\ & \mid \langle r_{in} \leftarrow m = when(e) e_1 \rangle \mid \langle v \leftarrow m = when(e) r_{in} \rangle \end{aligned}$$

Top Reduction Contexts:

$$r_{top} \ ::= \ gs(r_{in}, a) \mid ret(r_{in}, a) \mid ga(r_{in}, c) \mid nonret(r_{in})$$

The reduction contexts identify which subexpression of a given expression is to be evaluated next. These contexts correspond to the standard call-by-value reduction strategy. Because we have two forms of expressions, we need two forms of reduction contexts: top and inner. The following lemma tells us that local computation inside objects is deterministic.

Lemma 1 (Unique Decomposition) *Given an expression te, then either te is a top value or there exists a unique (r_{top}, e_{rdx}) such that $te = r_{top}[e_{rdx}]$.*

4.2 Reduction Rules

In this section we define the transition relation $\longmapsto$ between configurations. We describe in detail the rules for evaluating asynchronous, non-self-inflicted invocations and method override because these rules illustrate the key concepts of the transition system. The other rules are similar to these or are straightforward and can be found in Appendix A. Although this transition relation describes an interleaving semantics for the calculus, a straightforward modification produces a truly concurrent semantics.

Asynchronous, Non-Self-Inflicted Reductions. These rules describe the computation that results when an object a sends object b the message m asynchronously. The first rule says that the execution of an asynchronous method invocation proceeds by putting the message in the pending queue. The constant nil is returned to a to reflect the fact that the message has been placed in the queue. Integer i serves as the name or index of the pending message.

$(\Leftarrow_a)$ *non-self-inflicted*

$$\left\langle\!\!\left\langle\ \alpha, (a, \eta, [r_{top}[b \Leftarrow_a m(v)]]) \mid \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (a, \eta, [r_{top}[nil]]) \mid \mu, i : b \Leftarrow_a m(v)\ \right\rangle\!\!\right\rangle$$

where i is fresh.

When object b is in the idle state and there is a pending message m in μ for b, then b evaluates its guard for m within top-level expression ga according to the rule (μ):

(μ)
$$\left\langle\!\!\left\langle\ \alpha, (b, \eta, [I]) \mid i : b \Leftarrow_a m(v'), \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (b, \eta, [ga(e\, b, i)]) \mid i : b \Leftarrow_a m(v'), \mu\ \right\rangle\!\!\right\rangle$$

where $\eta(m) = when(e)v$.

The top-level expression ga stores the index of the message send from the queue. This information is needed so that the proper message send will be removed from the queue when the method body starts its evaluation.

If the guard evaluates to $true$, rule (μ-$true$) directs object b to start evaluating the method body for m within the top-level expression $nonret$. This rule also removes the message send at index i from the queue, since that message send is now being executed. This rule shows the role of the top-level expressions. Indeed, if we simply had value $true$ not wrapped by ga as state of object b, we would not have enough information to figure out to which state b should now go.

(μ-$true$)

$$\left\langle\!\!\left\langle\ \alpha, (b, \eta, [ga(true, i)]) \mid i : b \Leftarrow_a m(v'), \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (b, \eta, [nonret(v\, b\, v')]) \mid \mu\ \right\rangle\!\!\right\rangle$$

where $\eta(m) = when(e)v$.

Finally, when the body of the method has been evaluated to a value, rule ($nonret$) returns object b to the idle state and throws away the resulting value. Note that, similar to above, if we simply had value v'' as the state of object b, we would not have enough information to determine that v'' is the result of an asynchronous method invocation, and hence should be thrown away.

$(nonret)$
$$\left\langle\!\!\left\langle\ \alpha, (b, \eta, [nonret(v'')]) \mid \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (b, \eta, [I]) \mid \mu\ \right\rangle\!\!\right\rangle$$

Method Override Reductions. There are two different rules for evaluating the method override operation, one for self-inflicted and one for non-self-inflicted operations. In the self-inflicted case, we simply replace the guard and body of the method m in the method table of a:

$(\leftarrow)$ *self-inflicted*

$$\left\langle\!\!\left\langle\ \alpha, (a, \eta, [r_{top}[\langle a \leftarrow m = when(e)v\rangle]]) \mid \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (a, \eta[m]{:}= when(e)v, [r_{top}[a]]) \mid \mu\ \right\rangle\!\!\right\rangle$$

Note that because this operation is self-inflicted, we do not wait for a to become idle before performing the update. The notation $\eta[m]{:}= when(e)v$ stands for the function η' that is just like η except that it maps m to $when(e)v$. This operation returns the address of the modified object as its result. In the non-self-inflicted case, we must wait for b to enter the idle state before performing a method override requested by object a.

($\leftarrow$) *non-self-inflicted*

$$\langle\!\langle\ \alpha, (a, \eta, [r_{top}[\langle b \leftarrow m = when(e)v \rangle]]), (b, \eta', [I]) \mid \mu\ \rangle\!\rangle \longmapsto$$
$$\langle\!\langle\ \alpha, (a, \eta, [r_{top}[b']]), (b, \eta', [I]), (b', \eta'[m] := when(e)v, [I]) \mid \mu\ \rangle\!\rangle$$

where b' is a fresh address.

This rule first clones object b to create a new object b' and then replaces b''s m method with the new guard and body. The address of the new object is returned. Note that we cannot clone object b if it is busy. To see this point, suppose we create b' while b is executing e. If we put b' in the idle state, we can violate the integrity of b', because we do not complete the pending computation e, which could be responsible for restoring some invariant for b'. On the other hand, if we clone the state of b as well, we execute the pending computation e twice, potentially causing unwanted side-effects.

5 Type and Effect System

For the most part, the type system presented here is similar to the one defined in [FHM94]. The most novel parts are the effect system and the rules for typing asynchronous method invocation and guards.

5.1 Pro Types

The type of an object is called a *pro type*, short for *prototype*. The following type expression:

$$prot.\langle\!\langle m_1 : \tau_1 \rightarrow \tau_1', \ldots, m_k : \tau_k \rightarrow \tau_k' \rangle\!\rangle$$

defines a type t with the property that any expression e of this type is an object such that for $1 \leq i \leq k$, the result of $e \Leftarrow_s m_i(e_i)$ is a value of type τ_i', if e_i is of type τ_i. Keyword *pro* is a type binding operator. When bound type variable t appears in the types $\tau_1 \ldots \tau_k$, $\tau_1' \ldots \tau_k'$, it refers to the entire type. Thus, when we say $e \Leftarrow_s m_i(e_1)$ has type τ_i', we mean type τ_i' with any free occurrences of t in τ_i' replaced by the type $prot.\langle\!\langle m_1 : \tau_1 \rightarrow \tau_1', \ldots, m_k : \tau_k \rightarrow \tau_k' \rangle\!\rangle$. Thus, *pro types* are a special form of recursive type.

As an example of this kind of type, we may give the one-slot buffer object considered above the type:

$$\begin{array}{lll} buff : prot.\langle\!\langle & x : unit \rightarrow int, & size : unit \rightarrow int, \\ & put : int \rightarrow t, & get : unit \rightarrow int \rangle\!\rangle \end{array}$$

5.2 Effect, Types, Rows and Kinds

Our static analysis includes an effect system [LG88] that ensures that the evaluation of guards is side-effect free. This guarantee is important because the guard for a method may be invoked any number of times before its corresponding body is allowed to proceed. Since this number is a property of how the system orders pending messages, it is undesirable for guards to produce any observable effects.

We formalize this requirement by adding *pure* and *impure* effects to our static analysis. Only those expressions with *pure* effect will be permitted in guards because such expressions are guaranteed to produce no observable effects. Of course, a finer analysis of effects is possible; however for our current purposes, this coarse division suffices. The effect expressions include the constants *pure* and *impure*, and $\epsilon_1 \vee \epsilon_2$, which is impure if either ϵ_1 or ϵ_2 is impure.

Effects

$$\epsilon \ ::= \ pure \,|\, impure \,|\, \epsilon_1 \vee \epsilon_2$$

The type expressions include type variables, function types, *pro* types, and the constant types *bool*, *unit* and *int*.

Types

$$\tau \ ::= \ t \,|\, \tau_1 \xrightarrow{\epsilon} \tau_2 \,|\, pro\, t\,\mathbf{.}\, R \,|\, bool \,|\; unit \,|\; int$$

A behavior σ consists of a type and an effect.

Behavior

$$\sigma \ ::= \ \tau \,\&\, \epsilon$$

The row expressions appear as subexpression of type expressions, with row and types distinguished by kinds. Intuitively, the elements of kind $\{\vec{m}\}$ are rows that do not include method names $\{\vec{m}\}$. The reason we must know statically that some method does not appear is to guarantee that methods are not multiply defined. Kinds of the form $T \rightarrow \{\vec{m}\}$ are used to infer a form of higher-order polymorphism of method guards and bodies.

Rows

$$R \ ::= \ r \,|\, \langle\!\langle\rangle\!\rangle \,|\, \langle\!\langle R \,|\, m : \sigma \rangle\!\rangle \,|\; \lambda t.\, R \,|\, R\tau$$

Kinds

$$\begin{array}{lll} kind & ::= & T \,|\, \kappa \\ \kappa & ::= & \{\vec{m}\} \,|\, T \rightarrow \{\vec{m}\} \end{array}$$

The contexts of the system list term, type, and row variables.

Contexts

$$\Gamma \ ::= \epsilon \,|\, \Gamma\,, x : \tau \,|\, \Gamma\,, t : T \,|\, \Gamma\,, r : \kappa$$

The judgment forms are the following:

$\Gamma \vdash *$	well-formed context
$\Gamma \vdash e : \sigma$	term has behavior σ
$\Gamma \vdash \tau : T$	well-formed type
$\Gamma \vdash R : \kappa$	row has kind

5.3 Typing and Effecting Rules

To give the intuition for the effect system, we consider the rule for typing functional expressions:

$$(exp\ abs) \qquad \frac{\Gamma,\, x:\tau_1 \vdash e : \tau_2 \ \&\ \epsilon}{\Gamma \vdash \lambda x.\, e : \tau_1 \xrightarrow{\epsilon} \tau_2 \ \&\ pure}$$

The lambda abstraction by itself is pure because it is a value, but we must keep track of the *latent* effect of its body e. This effect will be visible when we apply the lambda abstraction to an argument. Hence, we annotate the function type with the latent effect.

The typing rule for sending a message asynchronously is the following:

$$(pro \Leftarrow_a\ asynch) \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \ \&\ \epsilon_1 \\ \Gamma \vdash e_2 : [\tau/t]\tau_1 \ \&\ \epsilon_2\end{array}}{\Gamma \vdash e_1 \underset{a}{\Leftarrow} m(e_2) : unit \ \&\ impure}$$

$$\text{where } \tau = prot.\langle\!\langle R \,|\, m : (\tau_1 \xrightarrow{\epsilon} \tau_2) \ \&\ \epsilon' \rangle\!\rangle$$

The first hypothesis of this rule requires e_1 to be an object that has at least a method m with type $\tau_1 \to \tau_2$. The second hypothesis forces the type of the parameter e_2 to be the same as the argument type of the method m, once we have substituted τ for any free variables t in τ_1. This substitution reflects the recursive structure of *pro* types. The type of the expression $e_1 \Leftarrow_a m(e_2)$ is *unit*, because the asynchronous method invocation does not return any result. The effect is impure because we modify the queue of pending messages.

Latent effects, similar to those in (*exp abs*), occur within methods. We discuss the effect portions of ($pro \Leftarrow_a$ *asynch*) in detail to illustrate how we track such effects. Because of call-by-value semantics, the body of the method m has the syntactic form $\lambda self.\, e$. So ϵ' records the latent effect of e. This effect is the one visible when we first reduce the method body by applying it to the host object and a parameter p: $(\lambda self.\, e)\, a\, p$. In other words, the expression $([a/self]e\,)p$ reduces to $(\lambda arg.\, e')\, p$ producing an effect ϵ'. Then $[p/arg]e'$ reduces to a value producing the latent effect ϵ.

The most complicated rule of the system is the (*pro ext*) rule:

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : prot.\langle\!\langle R \,|\, \vec{\ell} : \vec{\sigma} \rangle\!\rangle \ \&\ \epsilon_1 \\ \Gamma,\, t:T \vdash R : \{\vec{\ell}, m\} \\ \Gamma,\, r : T \to \{\vec{\ell},\, m\} \vdash e_2 : prot.\langle\!\langle r\,t \,|\, \vec{\ell} : \vec{\sigma},\, m : (\tau_1 \xrightarrow{\epsilon} \tau_2) \ \&\ \epsilon' \rangle\!\rangle \xrightarrow{pure} bool \ \&\ pure \\ \Gamma,\, r : T \to \{\vec{\ell},\, m\} \vdash e_3 : [prot.\langle\!\langle r\,t \,|\, \vec{\ell} : \vec{\sigma},\, m : (\tau_1 \xrightarrow{\epsilon} \tau_2) \ \&\ \epsilon' \rangle\!\rangle/t](t \xrightarrow{\epsilon'} \tau_1 \xrightarrow{\epsilon} \tau_2) \ \&\ \epsilon_3\end{array}}{\Gamma \vdash \langle e_1 \longleftrightarrow m{=}when(e_2)e_3 \rangle : prot.\langle\!\langle R \,|\, \vec{\ell} : \vec{\sigma},\, m : (\tau_1 \xrightarrow{\epsilon} \tau_2) \ \&\ \epsilon' \rangle\!\rangle \ \&\ impure}$$

In this rule, the first two hypotheses together require that e_1 is an expression with a *pro* type that does not include a method m, the method to be added. The last two assumptions are typings for e_2 and e_3, the expressions to be used respectively as the guard and body for m. The first thing to notice about the typing for e_3 is that it contains a row variable r, which is implicitly universally quantified. Because of this quantification, e_3 will have the indicated type for any substitution of row expression R for r, provided R has the correct kind. This is essential, since it implies that e_3 will have the required functionality for any possible future extension of $\langle e_1 \longleftrightarrow m{=}when(e_2)e_3\rangle$. The second important property of the typing of e_3 is that the type has the form $t \xrightarrow{\epsilon'} \tau_1 \xrightarrow{\epsilon} \tau_2$, with the extended *pro* type substituted for t. While t is hidden in the *pro* type of $\langle e_1 \longleftrightarrow m{=}when(e_2)e_3\rangle$, it is necessary in the hypothesis since sending the message m to $\langle e_1 \longleftrightarrow m{=}when(e_2)e_3\rangle$ will result in the application of e_3 to the extended object. The structure of the effects for e_3 is the same as described above, except it includes an additional effect ϵ_3, to reflect the effect of reducing e_3 to a value. The typing assumptions for the guard e_2 says that it can take any future extension of $\langle e_1 \longleftrightarrow m{=}when(e_2)e_3\rangle$ as its self parameter and that it returns a boolean value. The fact that each guard is a function from a self parameter to a boolean means that guards can access various components of their host objects. The effect assumptions for e_2 insure that both the immediate and the latent effects for the guard are pure.

The rule for method override has the same form as (*pro ext*). The other rules are straightforward and appear in Appendix B.

6 Main Results

In this section we present subject reduction, side-effect freeness for guards, and type soundness theorems. We prove these results for programs, which are closed, address-free terms typeable in the empty context. Given a program e, its possible computations originate from the following *initial configuration*:

$$\langle\!\langle\, (main, \eta, [I]) \mid \mu \,\rangle\!\rangle \qquad \text{where } \eta(begin) = \lambda self.\, \lambda arg.\, e \text{ and } \mu(1) = main \Leftarrow_a begin$$

which contains one object, whose address is *main*, with a method *begin* storing the program and one message pending invoking method *begin* of *main*.

We let g and its decorated variants range over configurations. A *computation sequence* is a finite sequence of transitions of the form $[g_i \longmapsto g_{i+1} \mid i < n]$, for some natural number n, where g_0 is an initial configuration.

6.1 Subject Reduction

Using techniques similar to those of [Har94], we prove a subject reduction theorem by extending typing judgments to type object addresses and method tables. With this intent, we extend our contexts to contain typing assumptions for object addresses:

$$\Gamma \ ::= \ \ldots \mid \Gamma, a : \tau$$

where τ is a closed *pro* type. We use Λ to indicate the projection of the context Γ onto the set of object addresses A. More formally, $a : \tau \in \Lambda$ iff $a : \tau \in \Gamma$ and $a \in A$. To state our subject reduction theorem, we need the following definitions.

Definition 1 *A method table η_a of an object a is typeable in Γ if when $\Lambda(a) = \mathbf{prot}.\langle\!\langle R \mid \vec{m} : \vec{\tau} \,\&\, \vec{\epsilon}\rangle\!\rangle$ and $\forall m_i \in dom(\eta_a)$, $1 \le i \le k$, $\eta_a(m_i) = when(e_i)v_i$, then the judgments*

$\Gamma, r : T \to \{\vec{m}\} \vdash v_i : [prot.\langle\!\langle r\, t \mid \vec{m} : \vec{\tau} \,\&\, \vec{\epsilon} \rangle\!\rangle/t](t \xrightarrow{\epsilon_i} \tau_i) \,\&\, pure$

and

$\Gamma, r : T \to \{\vec{m}\} \vdash e_i : prot.\langle\!\langle r\, t \mid \vec{m} : \vec{\tau} \,\&\, \vec{\epsilon} \rangle\!\rangle \xrightarrow{pure} bool \,\&\, pure,$

are both derivable.

In this definition $\vec{m} : \vec{\tau} \,\&\, \vec{\epsilon}$ is an abbreviation for $m_1 : \tau_1 \,\&\, \epsilon_1, \ldots, m_k : \tau_k \,\&\, \epsilon_k$. Note that the types of guards and bodies in the method table contain a free row variable. This reflects the fact that method bodies and guards may be used in extended objects, so they must have the required functionality for any possible future extension of their current host object.

In the next definitions $ObjAddr(\alpha)$ stands for the set of addresses of objects contained in α.

Definition 2 *An object soup α is typeable in Γ if $dom(\Lambda)$=$ObjAddr(\alpha)$ and $\forall a \in ObjAddr(\alpha)$, the method table η_a is typeable in Γ and if a is busy with state e_a, there exist type τ_a and effect ϵ_a such that the judgment $\Gamma \vdash e_a : \tau_a \,\&\, \epsilon_a$ is derivable.*

Definition 3 *A pending queue μ is typeable in Γ if $\forall i \in dom(\mu)$, there exist type τ_i and effect ϵ_i such that the judgment $\Gamma \vdash \mu(i) : \tau_i \,\&\, \epsilon_i$ is derivable.*

Definition 4 *A configuration $\langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle$ is typeable in Γ if α and μ are typeable in Γ.*

In proving subject reduction, we are not interested in showing that the types of top-level expressions are preserved by reduction. Indeed, the types of these expressions are trivially preserved as they are always *unit* (see the typing rules in the Appendix B). Instead, we are interested in preserving the types of the expressions that occur one level below the top expressions. For example, if the state of object a is $e_a \equiv ret(r_{in}[e_{rdx}], b)$ then $r_{in}[e_{rdx}]$ is a's one-level-down expression. If after one reduction step we have $e'_a \equiv ret(r_{in}[e'], b)$, then we want to prove that $r_{in}[e']$ has whatever type we gave to $r_{in}[e_{rdx}]$. Moreover, note that subject reduction holds only for objects whose states before and after a transition are related. In particular, if an object passes through an idle state or transitions from evaluating a guard to evaluating a method body, then the types of its state before and after the transition will be unrelated. This makes sense, since the computations before and after such a transition are not connected.

Definition 5 *If $\langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle \longmapsto \langle\!\langle\ \alpha' \mid \mu'\ \rangle\!\rangle$ via some transition rule T with $a \in ObjAddr(\alpha)$, we say that the states of a in α and α' are related if a is busy in α and α', and T is neither (μ-true) nor ($\Leftarrow_s$-true) on object a.*

We order effects as follows: $pure \leq impure$. The notation $\epsilon' \leq \epsilon$ means that if ϵ is *pure* then ϵ' is *pure* as well.

Definition 6 *Given a transition $g \longmapsto g'$ with $g = \langle\!\langle\ \alpha \mid \mu\ \rangle\!\rangle$ and $g' = \langle\!\langle\ \alpha' \mid \mu'\ \rangle\!\rangle$ typeable in Γ and Γ' respectively, we say that g and g' are compatible if:*

- *α and α' are compatible, that is, for all objects a whose states e_a and e'_a in α and α' are related, there exist type τ_a and effects ϵ_a and ϵ'_a such that the judgments $\Gamma \vdash r_{in}[e_{rdx}]_a : \tau_a \,\&\, \epsilon_a$*[4] *and $\Gamma' \vdash r_{in}[e_{rdx}]'_a : \tau_a \,\&\, \epsilon'_a$, with $\epsilon'_a \leq \epsilon_a$ are both derivable;*

- *μ and μ' are compatible, that is, $\forall\, i \in dom(\mu) \cap dom(\mu')$, there exists type τ and effect ϵ such that the judgments $\Gamma \vdash \mu(i) : \tau \,\&\, \epsilon$ and $\Gamma' \vdash \mu'(i) : \tau \,\&\, \epsilon$ are derivable.*

Theorem 1 (Subject Reduction) *Given a computation sequence $[g_i \longmapsto g_{i+1} \mid i < n]$, if g_0 is typeable in some context Γ_0, then $\forall i < n$, there exist contexts Γ, Γ_i, Γ_{i+1}, with $\Gamma_{i+1} \equiv \Gamma_i, \Gamma$, such that g_i and g_{i+1} are typeable in Γ_i and Γ_{i+1} respectively and are compatible.*

Side-effect freeness for guards and type soundness follow as corollaries of the previous theorem under the same hypothesis.

Corollary 1 (Effect Freeness) *we have that guard evaluation is side-effect free.*

Definition 7 *We define the error expressions of an object soup α to be those expressions of the forms (where r_c can be either r_{in} or r_{top}):*

- *$r_c[v_1\, v_2]$ where $v_1 \neq \lambda x.e$ for some e;*

- *$r_c[v_1 \Leftarrow m(v_2)]$ where v_1 is not an object address a such that $a \in ObjAddr(\alpha)$ and $\eta_a(m)$ exists.*

- *$r_c[v_1 \leftarrow m = when(e)v_2]$ where v_1 is not an object address a such that $a \in ObjAddr(\alpha)$ and $m \in dom(\eta_a)$.*

- *$r_c[v_1 \longleftarrow\!\!+\; m = when(e)v_2]$ where v_1 is not an object address a such that $a \in ObjAddr(\alpha)$ and $m \notin dom(\eta_a)$.*

Corollary 2 (Type Soundness) *Given a computation sequence $[g_i \longmapsto g_{i+1} | i < n]$, if g_0 is typeable in some context Γ_0, then for every configuration $g_i = \left\langle\!\!\left\langle\; \alpha_i \mid \mu_i \;\right\rangle\!\!\right\rangle$ and for every busy object $a \in ObjAddr(\alpha_i)$, e_a is not an error expression of the object soup α_i.*

Type soundness, which follows from Theorem 1, guarantees that the type system statically detects all expressions that can reduce to the following error expressions: applying a non-functional value to an argument, sending an object a message for which it has no defined method (*message not understood* error), overriding a method which has not been defined, and extending an object with a method it already has.

[4]Note that if e_a is typeable in Γ, then its one-level down expression $r_{in}[e_{rdx}]_a$ is typeable in Γ as well.

7 Conclusions

We have presented what we believe is the first typed, prototype-based calculus for concurrent objects. We have described an operational semantics using a transition system between configurations and have given a type and effect system. We have proven the soundness of our static analysis with respect to the operational semantics via a subject reduction theorem.

This work is intended as a starting point for studying the theoretical foundations of concurrent object-oriented programming. In the following, we briefly describe some of issues we intend to investigate further. Our current type system does not support subtyping because subtyping is unsound in pure prototype-based calculi [FM94]. This problem has been solved for the sequential version of our calculus [FM95], and we believe this solution will carry over to our concurrent setting. A second research direction focuses on method availability. Our type soundness theorem demonstrates that the type system we have given detects "message not understood" errors at compile time, in the sense that no object will ever receive a message for which it has no method defined. However, the theorem does not ensure that the method in question is available (by virtue of having a *true* guard) when its execution is required. Although method unavailability is not a problem for asynchronous method invocation, it can cause deadlock in the synchronous case. Addressing this problem requires additional analysis to account for the communication behavior of objects. Such analyses may be done by modeling communication behaviors as process algebra expressions in the style of [NN93, Nie93]. Finally, we would like to investigate the equational theory of our calculus by defining an observational semantics in the style of [AMS92].

Acknowledgment We are grateful to Carolyn Talcott for insightful discussions and carefully reading a draft of this paper.

References

[AC94] M. Abadi and L. Cardelli. A theory of primitive objects: untyped and first order systems. In *Proc. of TACS'94*, 1994.

[AC95] M. Abadi and L. Cardelli. An imperative object calculus: basic typing and soundness. In *Proc. of second ACM-SIGPLAN workshop on state in programming Languages*, 1995.

[Agh86] G. Agha. Actors: a model of concurrent computation in distributed systems. *MIT Press*, Cambridge, Mass., 1986.

[Agh90] G. Agha. Concurrent object-oriented programming. In *Communication ACM* 33(9), 1990.

[Ame89] P. America. Issue in the design of a parallel object-oriented language. In *Formal Aspects of Computing*, 1, 366-411, 1989.

[AMS92] G. Agha, I. Mason, S. Smith, and C. Talcott. Towards a theory of actor computation. In *Proc. of CONCUR'92*, 1992.

[Car93] D. Caromel. Towards a method of concurrent object-oriented programming. In *Communication of ACM*, 36(9), 1993.

[Car95] L. Cardelli. A language with distributed scope. In *Computing Systems*, 8(1):27–59, 1995.

[CGH89] R. Chandra, A. Gupta, and J. Hennessy. COOL: A language for parallel programming. In *Proc. 2nd workshop on programming languages and compilers for parallel computing*, IEEE CS, 1989.

[FA93] S. Frolund and G. Agha. A language framework for multi-object coordination. In *Proc. of ECOOP'93*, 1993.

[FHM94] K. Fisher, F. Honsell, and J.C. Mitchell. A lambda calculus of objects and method specialization. In *Nordic J. Computing (*formerly *BIT)*, 1:3–37, 1994. Preliminary version appeared in *Proc. IEEE Symp. on Logic in Computer Science,*, 26–38, 1993.

[FM94] K. Fisher and J.C. Mitchell. Notes on typed object-oriented programming. In *Proc. Theoretical Aspects of Computer Software*, pages 844–885. Springer LNCS 789, 1994.

[FM95] K. Fisher and J.C. Mitchell. A delegation-based object calculus with subtyping. In *Proc. 10th Int'l Conf. Fundamentals of Computation Theory (FCT'95)*, pages 42–61. Springer LNCS 965, 1995.

[Har94] R. Harper. A simplified account of polymorphic references. In *Information Processing Letters* 51(4), 1994.

[Jon93] C. Jones. A pi-calculus semantics for an object-based design notation. In *Proc. of CONCUR'93*, 1993.

[KY94] N. Kobayashi and A. Yonezawa. Type theoretic foundations for object-oriented concurrent programming. In *Proc. of OOPSLA'94*, 1994.

[LG88] J. Lucassen and D. Gifford. Polymorphic effect systems. In *Proc. of POPL'88*, 1988.

[MY93] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.

[Nie92] O. Nierstrasz. Towards an object calculus. In *Proc. of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, 1992.

[Nie93] O. Nierstrasz. Regular types for active objects. In *Proc. of OOPSLA'93*, 1993.

[NN93] F. Nielson and H. Nielson. From CML to process algebra. In *Proc. of CONCUR'93*, 1993.

[PT94] B. Pierce and D. Turner. Concurrent objects in a process calculus. In *Proc. of Theory and Practice of Parallel Programming*, 1994.

[Tal96] C. Talcott. Interaction semantics for components of distributed systems. In *Proc. of FMOODS96*, 1996.

[Vas94] V. T. Vasconcelos. Typed concurrent objects. In *Proc. ECOOP'94*, 1994.

[WKH92] B. Wyatt, K. Kavi, and S. Hufnagel. Parallelism in object-oriented languages: A survey. In *IEEE Software*, 1992.

[Yon90] A. Yonezawa. ABCL: An object-oriented concurrent system. *MIT Press*, Cambridge, Mass., 1990.

A Operational Semantics

(λ) $\quad \left\langle\!\!\left\langle\; \alpha, (a, \eta, [r_{top}[(\lambda x.\, e)v]]) \mid \mu \;\right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\; \alpha, (a, \eta, [r_{top}[[v/x]e]]) \mid \mu \;\right\rangle\!\!\right\rangle$

$(\langle\rangle)$ $\quad \left\langle\!\!\left\langle\; \alpha, (a, \eta, [r_{top}[\langle\rangle]]) \mid \mu \;\right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\; \alpha, (a, \eta, [r_{top}[a']]), (a', \eta', [I]) \mid \mu \;\right\rangle\!\!\right\rangle$

where $dom(\eta') = \emptyset$ and $a' \notin ObjAddr(\alpha)$.

$(\leftarrow)$ *self-inflicted*

$$\left\langle\!\!\left\langle\; \alpha, (a, \eta, [r_{top}[\langle a \leftarrow m = when(e)v\rangle]]) \mid \mu \;\right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\; \alpha, (a, \eta[m]{:=}\, when(e)v, [r_{top}[a]]) \mid \mu \;\right\rangle\!\!\right\rangle$$

$(\leftarrow)$ *non-self-inflicted*

$$\left\langle\!\!\left\langle\; \alpha, (a, \eta, [r_{top}[\langle b \leftarrow m = when(e)v\rangle]]), (b, \eta', [I]) \mid \mu \;\right\rangle\!\!\right\rangle \longmapsto$$
$$\left\langle\!\!\left\langle\; \alpha, (a, \eta, [r_{top}[b']]), (b, \eta', [I]), (b', \eta'[m]{:=}\, when(e)v, [I]) \mid \mu \;\right\rangle\!\!\right\rangle$$

where $b' \notin ObjAddr(\alpha)$.

$(\leftarrow\!\!\!+)$ *non-self-inflicted*

$$\left\langle\!\!\left\langle\; \alpha, (a, \eta, [r_{top}[\langle b \leftarrow\!\!\!+\; m = when(e)v\rangle]]), (b, \eta', [I]) \mid \mu \;\right\rangle\!\!\right\rangle \longmapsto$$
$$\left\langle\!\!\left\langle\; \alpha, (a, \eta, [r_{top}[b']]), (b, \eta', [I]), (b', \eta'[m]{:=}\, when(e)v, [I]) \mid \mu \;\right\rangle\!\!\right\rangle$$

where $b' \notin ObjAddr(\alpha)$.

$(\Leftarrow_a)$ *self-inflicted*

$$\left\langle\!\!\left\langle\; \alpha, (a, \eta, [r_{top}[a \Leftarrow_a m(v')]]) \mid \mu \;\right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\; \alpha, (a, \eta, [r_{top}[(\lambda x.\, nil)(v\, a\, v')]]) \mid \mu \;\right\rangle\!\!\right\rangle$$

where $\eta(m) = when(e)v$.

$(\Leftarrow_a)$ *non-self-inflicted*

$$\left\langle\!\!\left\langle\; \alpha, (a, \eta, [r_{top}[b \Leftarrow_a m(v)]]) \mid \mu \;\right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\; \alpha, (a, \eta, [r_{top}[nil]]) \mid \mu, i : b \Leftarrow_a m(v) \;\right\rangle\!\!\right\rangle$$

where i is fresh.

The next four rules share the condition that $\eta(m) = when(e)v$.

(μ) $\quad \left\langle\!\!\left\langle\ \alpha, (a, \eta, [I]) \mid i : a \Leftarrow_a m(v'), \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (a, \eta, [ga(e\,a, i)]) \mid i : a \Leftarrow_a m(v'), \mu\ \right\rangle\!\!\right\rangle$

$(\mu\text{-}true)$ $\quad \left\langle\!\!\left\langle\ \alpha, (a, \eta, [ga(true, i)]) \mid i : a \Leftarrow_a m(v'), \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (a, \eta, [nonret(v\,a\,v')]) \mid \mu\ \right\rangle\!\!\right\rangle$

$(\mu\text{-}false)$ $\quad \left\langle\!\!\left\langle\ \alpha, (a, \eta, [ga(false, i)]) \mid i : a \Leftarrow_a m(v'), \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (a, \eta, [I]) \mid i : a \Leftarrow_a m(v'), \mu\ \right\rangle\!\!\right\rangle$

$(nonret)$ $\quad \left\langle\!\!\left\langle\ \alpha, (a, \eta, [nonret(v)]) \mid \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (a, \eta, [I]) \mid \mu\ \right\rangle\!\!\right\rangle$

$(\Leftarrow_s)$ *self-inflicted*

$$\left\langle\!\!\left\langle\ \alpha, (a, \eta, [r_{top}[a \Leftarrow_s m(v')]]) \mid \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (a, \eta, [r_{top}[v\,a\,v']]) \mid \mu\ \right\rangle\!\!\right\rangle$$

where $\eta(m) = when(e)v$.

The next four rules share the condition that $\eta_b(m) = when(e)v$.

$(\Leftarrow_s)$ *non-self-inflicted*

$$\left\langle\!\!\left\langle\ \alpha, (a, \eta_a, [r_{top}[b \Leftarrow_s m(v')]]), (b, \eta_b, [I]) \mid \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (a, \eta_a, [r_{top}[b \Leftarrow_s m(v')]]), (b, \eta_b, [gs(e\,b, a)]) \mid \mu\ \right\rangle\!\!\right\rangle$$

$(\Leftarrow_s\text{-}true)$

$$\left\langle\!\!\left\langle\ \alpha, (a, \eta_a, [r_{top}[b \Leftarrow_s m(v')]]), (b, \eta_b, [gs(true, a)]) \mid \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (a, \eta_a, [r_{top}[b \Leftarrow_s m(v')]]), (b, \eta_b, [ret(v\,b\,v', a)]) \mid \mu\ \right\rangle\!\!\right\rangle$$

$(\Leftarrow_s\text{-}false)$

$$\left\langle\!\!\left\langle\ \alpha, (b, \eta_b, [gs(false, a)]) \mid \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (b, \eta_b, [I]) \mid \mu\ \right\rangle\!\!\right\rangle$$

$(\Leftarrow_s\text{-}ret)$

$$\left\langle\!\!\left\langle\ \alpha, (a, \eta_a, [r_{top}[b \Leftarrow_s m(v')]]), (b, \eta_b, [ret(v'', a)]) \mid \mu\ \right\rangle\!\!\right\rangle \longmapsto \left\langle\!\!\left\langle\ \alpha, (a, \eta_a, [r_{top}[v'']]), (b, \eta_b, [I]) \mid \mu\ \right\rangle\!\!\right\rangle$$

B Type and Effect Rules

Context Rules

$(start\ \Gamma)$

$$\frac{}{\epsilon \vdash *}$$

$(type\ var)$

$$\frac{\Gamma \vdash * \qquad t \notin dom(\Gamma)}{\Gamma, t : T \vdash *}$$

$(row\ var)$

$$\frac{\Gamma \vdash * \qquad r \notin dom(\Gamma)}{\Gamma,\ r : T \to \{l_1, \ldots, l_k\} \vdash *}$$

(*exp var*)

$$\frac{\Gamma \vdash \tau : T \quad x \notin dom(\Gamma)}{\Gamma,\, x : \tau \vdash *}$$

(*exp addr*)

$$\frac{\Gamma \vdash \tau : T \quad a \notin dom(\Gamma)}{\Gamma, a : \tau \vdash *}$$

Rules for type expressions

(*type const*)

$$\frac{\Gamma \vdash *}{\Gamma \vdash \theta : T}$$

where θ can be *bool*, *unit* or *int*.

(*type projection*)

$$\frac{\Gamma \vdash * \quad t : T \in \Gamma}{\Gamma \vdash t : T}$$

(*type arrow*)

$$\frac{\Gamma \vdash \tau_1 : T \quad \Gamma \vdash \tau_2 : T}{\Gamma \vdash \tau_1 \xrightarrow{\epsilon} \tau_2 : T}$$

(*type pro*)

$$\frac{\Gamma, t : T \vdash R : \{m_1, \ldots, m_k\}}{\Gamma \vdash \mathit{pro}\, t.R : T}$$

Rules for rows

(*empty row*)

$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\!\langle\rangle\!\rangle : \{m_1, \ldots, m_k\}}$$

(*row projection*)

$$\frac{\Gamma \vdash * \quad r : \kappa \in \Gamma}{\Gamma \vdash r : \kappa}$$

(*row label*)

$$\frac{\Gamma \vdash R : T^i \to \{m_1, \ldots, m_k\} \quad \{n_1, \ldots, n_\ell\} \subseteq \{m_1, \ldots, m_k\} \quad i \in \{0, 1\}}{\Gamma \vdash R : T^i \to \{n_1, \ldots, n_\ell\}}$$

$$\frac{\Gamma \vdash R : \{m, m_1, \ldots, m_k\} \quad \Gamma \vdash \tau : T}{\Gamma \vdash \langle\!\langle R \mid m : \tau \;\&\; \epsilon \rangle\!\rangle : \{m_1, \ldots, m_k\}} \quad (row\ ext)$$

$$\frac{\Gamma, t : T \vdash R : \{m_1, \ldots, m_k\}}{\Gamma \vdash \lambda t.\, R : T \rightarrow \{m_1, \ldots, m_k\}} \quad (row\ fn\ abs)$$

$$\frac{\Gamma \vdash R : T \rightarrow \{m_1, \ldots, m_k\} \quad \Gamma \vdash \tau : T}{\Gamma \vdash R\tau : \{m_1, \ldots, m_k\}} \quad (row\ fn\ app)$$

Type and Row Equality

Type or row expressions that differ only in names of bound variables or order of *label* : *type* pairs are considered identical. In other words, we consider α-conversion of type variables bound by λ or *pro* and applications of the principle

$$\langle\!\langle \langle\!\langle R \mid n : \sigma_1 \rangle\!\rangle \mid m : \sigma_2 \rangle\!\rangle = \langle\!\langle \langle\!\langle R \mid m : \sigma_2 \rangle\!\rangle \mid n : \sigma_1 \rangle\!\rangle$$

within type or row expressions to be conventions of syntax, rather than explicit rules of the system. Additional equations arise as a result of β-reduction, written $\rightarrow_\beta$, or β-conversion, written $\leftrightarrow_\beta$.

$$\frac{\Gamma \vdash R : \kappa, \quad R \rightarrow_\beta R'}{\Gamma \vdash R' : \kappa} \quad (row\ \beta)$$

$$\frac{\Gamma \vdash \tau : T, \quad \tau \rightarrow_\beta \tau'}{\Gamma \vdash \tau' : T} \quad (type\ \beta)$$

$$\frac{\Gamma \vdash e : \tau \,\&\, \epsilon, \quad \tau \leftrightarrow_\beta \tau', \quad \Gamma \vdash \tau' : T}{\Gamma \vdash e : \tau' \,\&\, \epsilon} \quad (type\ eq)$$

Rules for assigning types to terms

$$\frac{\Gamma \vdash * \quad c : \sigma \in Const}{\Gamma \vdash c : \sigma} \quad (exp\ const)$$

$$\frac{\Gamma \vdash * \quad a : \tau \in \Gamma}{\Gamma \vdash a : \tau \;\&\; pure} \quad (addr\ projection)$$

$$\frac{\Gamma \vdash * \quad x : \tau \in \Gamma}{\Gamma \vdash x : \tau \;\&\; pure} \quad (exp\ projection)$$

$$(exp\ abs)\qquad \frac{\Gamma,\, x:\tau_1 \vdash e \,:\, \tau_2 \;\&\; \epsilon}{\Gamma \vdash \lambda x.\, e \,:\, \tau_1 \xrightarrow{\epsilon} \tau_2 \;\&\; pure}$$

$$(exp\ app)\qquad \frac{\begin{array}{c}\Gamma \vdash e_1 \,:\, \tau_1 \xrightarrow{\epsilon} \tau_2 \;\&\; \epsilon_1 \\ \Gamma \vdash e_2 \,:\, \tau_1 \;\&\; \epsilon_2\end{array}}{\Gamma \vdash e_1\, e_2 \,:\, \tau_2 \;\&\; \epsilon_1 \vee \epsilon_2 \vee \epsilon}$$

$$(empty\ object)\qquad \frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle \,:\, prot.\langle\!\langle\rangle\!\rangle \;\&\; impure}$$

$$(pro \Leftarrow_a\ asynch)\qquad \frac{\begin{array}{c}\Gamma \vdash e_1 \,:\, \tau \;\&\; \epsilon_1 \\ \Gamma \vdash e_2 \,:\, [\tau/t]\tau_1 \;\&\; \epsilon_2\end{array}}{\Gamma \vdash e_1 \underset{a}{\Leftarrow} m(e_2) \,:\, unit \;\&\; impure}$$

where $\tau = prot.\langle\!\langle R \,|\, m \,:\, (\tau_1 \xrightarrow{\epsilon} \tau_2) \;\&\; \epsilon' \rangle\!\rangle$

$$(pro \Leftarrow_s\ synch)\qquad \frac{\begin{array}{c}\Gamma \vdash e_1 \,:\, \tau \;\&\; \epsilon_1 \\ \Gamma \vdash e_2 \,:\, [\tau/t]\tau_1 \;\&\; \epsilon_2\end{array}}{\Gamma \vdash e_1 \Leftarrow_s m(e_2) \,:\, [\tau/t]\tau_2 \;\&\; \epsilon_1 \vee \epsilon_2 \vee \epsilon' \vee \epsilon}$$

where $\tau = prot.\langle\!\langle R \,|\, m \,:\, (\tau_1 \xrightarrow{\epsilon} \tau_2) \;\&\; \epsilon' \rangle\!\rangle$

$$(pro\ ext)\qquad \frac{\begin{array}{l}\Gamma \vdash e_1 \,:\, prot.\langle\!\langle R \,|\, \vec{\ell} : \vec{\sigma} \rangle\!\rangle \;\&\; \epsilon_1 \\ \Gamma,\, t:T \vdash R \,:\, \{\vec{\ell}, m\} \\ \Gamma,\, r \,:\, T \to \{\vec{\ell},\, m\} \vdash \\ \qquad\qquad e_2 \,:\, prot.\langle\!\langle r\,t \,|\, \vec{\ell} : \vec{\sigma},\, m \,:\, (\tau_1 \xrightarrow{\epsilon} \tau_2) \;\&\; \epsilon' \rangle\!\rangle \xrightarrow{pure} bool \;\&\; pure \\ \Gamma,\, r \,:\, T \to \{\vec{\ell},\, m\} \vdash \\ \qquad e_3 \,:\, [prot.\langle\!\langle r\,t \,|\, \vec{\ell} : \vec{\sigma},\, m \,:\, (\tau_1 \xrightarrow{\epsilon} \tau_2) \;\&\; \epsilon' \rangle\!\rangle/t](t \xrightarrow{\epsilon'} \tau_1 \xrightarrow{\epsilon} \tau_2) \;\&\; \epsilon_3\end{array}}{\Gamma \vdash \langle e_1 \longleftrightarrow m{=}when(e_2)e_3\rangle \,:\, prot.\langle\!\langle R \,|\, \vec{\ell} : \vec{\sigma},\, m \,:\, (\tau_1 \xrightarrow{\epsilon} \tau_2) \;\&\; \epsilon' \rangle\!\rangle \;\&\; impure}$$

$$(pro\ ov)\qquad \frac{\begin{array}{l}\Gamma \vdash e_1 \,:\, prot.\langle\!\langle R \,|\, \vec{\ell} : \vec{\sigma},\, m \,:\, (\tau_1 \xrightarrow{\epsilon} \tau_2) \;\&\; \epsilon' \rangle\!\rangle \;\&\; \epsilon_1 \\ \Gamma,\, r \,:\, T \to \{\vec{\ell},\, m\} \vdash \\ \qquad\qquad e_2 \,:\, prot.\langle\!\langle r\,t \,|\, \vec{\ell} : \vec{\sigma},\, m \,:\, (\tau_1 \xrightarrow{\epsilon} \tau_2) \;\&\; \epsilon' \rangle\!\rangle \xrightarrow{pure} bool \;\&\; pure \\ \Gamma,\, r \,:\, T \to \{\vec{\ell},\, m\} \vdash \\ \qquad e_3 \,:\, [prot.\langle\!\langle r\,t \,|\, \vec{\ell} : \vec{\sigma},\, m \,:\, (\tau_1 \xrightarrow{\epsilon} \tau_2) \;\&\; \epsilon' \rangle\!\rangle/t](t \xrightarrow{\epsilon'} \tau_1 \xrightarrow{\epsilon} \tau_2) \;\&\; \epsilon_3\end{array}}{\Gamma \vdash \langle e_1 \leftarrow m{=}when(e_2)e_3\rangle \,:\, prot.\langle\!\langle R \,|\, \vec{\ell} : \vec{\sigma},\, m \,:\, (\tau_1 \xrightarrow{\epsilon} \tau_2) \;\&\; \epsilon' \rangle\!\rangle \;\&\; impure}$$

Rules to assigning types to top level expressions

$(exp\ gs)$

$$\frac{\Gamma \vdash e : bool \ \&\ pure \qquad \Gamma \vdash a : \tau \ \&\ pure}{\Gamma \vdash gs(e, a) : unit \ \&\ pure}$$

$(exp\ ret)$

$$\frac{\Gamma \vdash e : \tau_1 \ \&\ \epsilon \qquad \Gamma \vdash a : \tau_2 \ \&\ pure}{\Gamma \vdash ret(e, a) : unit \ \&\ \epsilon \vee pure}$$

$(exp\ ga)$

$$\frac{\Gamma \vdash e : bool \ \&\ pure \qquad \Gamma \vdash c : int \ \&\ pure}{\Gamma \vdash ga(e, c) : unit \ \&\ pure}$$

$(exp\ nonret)$

$$\frac{\Gamma \vdash e : \tau \ \&\ \epsilon}{\Gamma \vdash nonret(e) : unit \ \&\ \epsilon}$$