Electronic Notes in Theoretical Computer Science 16 No. 3 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume16.html 17 pages

A Concurrent Object Calculus: Reduction and Typing

Andrew D. Gordon [1]

University of Cambridge Computer Laboratory
Cambridge, United Kingdom

Paul D. Hankin

University of Cambridge Computer Laboratory
Cambridge, United Kingdom

Abstract

We obtain a new formalism for concurrent object-oriented languages by extending Abadi and Cardelli's imperative object calculus with operators for concurrency from the π-calculus and with operators for synchronisation based on mutexes. Our syntax of terms is extremely expressive; in a precise sense it unifies notions of expression, process, store, thread, and configuration. We present a chemical-style reduction semantics, and prove it equivalent to a structural operational semantics. We identify a deterministic fragment that is closed under reduction and show that it includes the imperative object calculus. A collection of type systems for object-oriented constructs is at the heart of Abadi and Cardelli's work. We recast one of Abadi and Cardelli's first-order type systems with object types and subtyping in the setting of our calculus and prove subject reduction. Since our syntax of terms includes both stores and running expressions, we avoid the need to separate store typing from typing of expressions. We translate asynchronous communication channels and the choice-free asynchronous π-calculus into our calculus to illustrate its expressiveness; the types of read-only and write-only channels are supertypes of read-write channels.

1 Motivation

A great deal of software is coded in terms of concurrent processes and objects. The purpose of our work is to develop a new formalism for expressing, typing, and reasoning about computations based on concurrent processes and objects.

[1] Current affiliation: Microsoft Research.

Our concurrent object calculus $\mathbf{concς_m}$ consists of Abadi and Cardelli's imperative object calculus $\mathbf{impς}$ extended with primitives for parallel composition and for synchronisation via mutexes. Our work extends the analysis by Abadi and Cardelli [1] of object-oriented features to concurrent languages. At the heart of their work is a series of type systems able to express a great variety of object-oriented idioms. Given $\mathbf{concς_m}$, we may smoothly and soundly extend these type systems to accommodate concurrency.

There are by now many formalisms capable of encoding objects and concurrency. Support of Abadi and Cardelli's type systems is one distinctive feature of our calculus. Others are the following. Unlike most process calculi, the syntax of $\mathbf{concς_m}$ includes sequential composition of expressions that are expected to return results; there is no need to encode results in terms of continuations. Rather than reducing concurrent objects to other concepts, $\mathbf{concς_m}$ treats objects as primitive. Rather than introduce auxiliary notions of stores or configurations or labelled transitions, we directly describe the semantics of $\mathbf{concς_m}$ in terms of a reduction relation on expressions.

As evidence of the expressiveness of our calculus, we present an encoding of the asynchronous π-calculus. An extended version of this paper, available from the authors, includes more examples, as well as full definitions and full proofs. Here are our main technical results.

First, we describe a semantics for concurrent objects based on a reduction relation and a structural congruence relation in the style of Milner's reduction semantics [16] for the π-calculus [17]. We prove that our reduction semantics is equivalent to a classical structural operational semantics defined using auxiliary notions of stores, threads, and configurations.

Second, we identify a single-threaded subset of our calculus that is preserved by reduction and includes the $\mathbf{impς}$-calculus.

Third, given a few simple rules for parallel composition and restriction, we confer Abadi and Cardelli's first-order type system with objects and subtyping, $\mathbf{Ob_{1<:}}$, on our calculus. We prove subject reduction for this system without needing any notion of store typing separate from the notion of expression typing.

1.1 Related work

Plotkin's structural operational semantics [22] is a standard technique for concurrent languages. A computation is described as a sequence of configurations. A configuration typically consists of a collection of runnable threads, a store, and other data such as the state of communication channels. Di Blasio and Fisher [8] describe their calculus of concurrent objects in this style. Other languages treated in this style include an actor language [2] and CML [4,23].

Ferreira, Hennessy, and Jeffrey [9] avoid configurations in their operational semantics for CML by employing a CCS-style labelled transition system. In their work, and in ours, the parallel composition $a \Rsh b$ of two expressions a

and b is an expression consisting of a and b running in parallel. Any result returned by b is returned by the whole composition; any result returned by a is discarded. So unlike the situation in most process calculi, parallel composition is not commutative: the effects of $a \Rsh b$ and $b \Rsh a$ are different. In implementation terms this is perfectly natural; running $a \Rsh b$ amounts to forking off a as a new thread and then running b.

Our reduction semantics is directly inspired by Milner's [16] presentation of the chemical abstract machine of Berry and Boudol [5]. In a chemical semantics, a computation state is represented by a term of the calculus; there is no need for the auxiliary notion of a configuration. Previous chemical semantics for concurrent languages use evaluation contexts to treat sequential composition of expressions [3,6,18]; instead, our semantics exploits a non-commutative parallel composition.

Di Blasio and Fisher's paper is the work most closely related to ours. Their principal results are the definition of a configuration-based reduction semantics for their calculus, a type soundness theorem, and the proof that certain guard expressions used for synchronisation have no side-effects. As in their work, we prove the soundness of a type system for concurrent objects. Our chemical semantics has no need for the auxiliary notions of configurations and reduction contexts used in theirs. Unlike their work, ours includes two independent but equivalent characterisations of our operational semantics.

Various formalisms in the π-calculus family have been used to model imperative or concurrent objects [7,10,12–14,20,24,25]. All these models use formalisms based on processes, computations with no concept of returning a result, instead of expressions. The operation of returning a result is translated using continuations into sending a message on a result channel. Our **concς**-calculus is based on expressions that return results because its precursor **impς** is based on expressions, because we do not wish to presuppose channel-based communication for returning results, and because expressions with results are a fundamental aspect of many programming languages and therefore deserve a semantics in their own right.

1.2 Organisation of the paper

In Section 2 we present the syntax and semantics of a core calculus of concurrent objects, the **concς**-calculus. In Section 3 we add mutexes to obtain the $\mathbf{conc\varsigma_m}$-calculus. Our syntax of terms unifies auxiliary notions of process, expression, store, and configuration, and hence supports a particularly simple reduction semantics. In Section 4 we show that our semantics corresponds precisely to a more conventional, but more complex, semantics phrased in terms of configurations. In Section 5 we demonstrate the soundness of the $\mathbf{Ob}_{1<:}$ type system for $\mathbf{conc\varsigma_m}$. Section 6 concludes the paper.

2 Concurrent Objects

We extend the imperative object calculus by adding names to objects, and adding parallel composition and name scoping operators from the π-calculus.

2.1 Syntax

We assume there are disjoint infinite sets of *names*, *variables*, and *labels*. We let p, q, and r range over names. We let x, y, and z range over variables. We let ℓ range over labels. We define the sets of *results*, *denotations*, and *terms* by the grammars:

Syntax of the concς-calculus

$u, v ::=$	results
$\quad x$	variable
$\quad p$	name
$d ::=$	denotations
$\quad [\ell_i = \varsigma(x_i)b_i \,^{i\in 1..n}]$	object
$a, b, c ::=$	terms
$\quad u$	result
$\quad p \mapsto d$	denomination
$\quad u.\ell$	method select
$\quad u.\ell \Leftarrow \varsigma(x)b$	method update
$\quad clone(u)$	cloning
$\quad let\ x{=}a\ in\ b$	let
$\quad a \Rsh b$	parallel composition
$\quad (\nu p)a$	restriction

In a method $\varsigma(x)b$, the variable x is bound; its scope is b. In a term $let\ x{=}a\ in\ b$, the variable x is bound; its scope is b. In a restriction, $(\nu p)a$, the name p is bound; its scope is a. Let $fn(a)$ and $fv(a)$ be the sets of names and variables, respectively, free in the term a. We write $a\{\!\{x \leftarrow v\}\!\}$ for the substitution of the result v for each free occurrence of x in term a. We write $a = b$ to mean that the terms a and b are equal up to the renaming of bound names and bound variables, and the reordering of the labelled components of objects.

Some syntactic conventions: $(\nu p)a \Rsh b$ is read $((\nu p)a) \Rsh b$, $u.\ell \Leftarrow \varsigma(x)b \Rsh c$ is read $(u.\ell \Leftarrow \varsigma(x)b) \Rsh c$, and $let\ x{=}a\ in\ b \Rsh c$ is read $(let\ x{=}a\ in\ b) \Rsh c$. We write $(\nu\vec{p})a$ for $(\nu p_1)(\nu p_2)\ldots(\nu p_n)a$ where $\vec{p} = p_1, p_2, \ldots, p_n$.

Our syntax distinguishes names, which represent the addresses of stored objects, from variables, which represent intermediate values. The distinction reflects the different uses of names and variables, but is not essential; we believe it will be useful when we come to treat observational equivalences. Results in our syntax are atomic names or atomic variables; our techniques would easily extend to structured results, such as tuples or λ-abstractions. We

obtained our syntax by directly combining that of the **impς**-calculus and the π-calculus. Our syntax uses separate constructs, restriction and denomination, for name scoping and name definition, respectively. This allows for cyclic dependencies between definitions. An alternative is to use a single construct defining several names simultaneously with mutually recursive scopes, as in the join-calculus [10] for example. Due to the generality of our syntax, we need a simple type system, defined in Section 4, to rule out certain terms as not well-formed. For example, a process such as $(p \mapsto [] \Rsh p \mapsto []) \Rsh p$, that contains two denominations for the same name, is not well-formed.

2.2 Informal Semantics

We may interpret a term of our object calculus either as a *process* or as an *expression*. A process is simply a concurrent computation. An expression is a concurrent computation that is expected to return a result. In fact, an expression may be regarded as a process, since we may always ignore any result that it returns.

A result u is an expression that immediately returns itself.

A denomination $p \mapsto [\ell_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$ is a process that confers the name p on the object $[\ell_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$. We say that the object $[\ell_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$ is the denotation of the name p. Intuitively, the process represents an object stored at a memory location and the name p represents the address of the object.

A method select $p.\ell$ is an expression that invokes the method labelled ℓ of the object denoted by p. In the presence of a denomination $p \mapsto [\ell_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$, where $\ell = \ell_j$ for some $j \in 1..n$, the effect of $p.\ell$ is to run the expression $b_j\{\!\{x_j \leftarrow p\}\!\}$, that is, to run the body b_j of the method labelled ℓ, with the variable x_j bound to the name of the object itself.

A method update $p.\ell \Leftarrow \varsigma(x)b$ is an expression that updates the method labelled ℓ of the object denoted by p. In the presence of a denomination $p \mapsto [\ell_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$, where $\ell = \ell_j$ for some $j \in 1..n$, the effect of $p.\ell \Leftarrow \varsigma(x)b$ is to update the denomination to be $p \mapsto [\ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i \ ^{i\in(1..n)-\{j\}}]$, and to return p as its result.

A clone $clone(p)$ is an expression that makes a shallow copy of the object denoted by p. In the presence of a denomination $p \mapsto [\ell_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$, the effect of $clone(p)$ is to generate a fresh name q with denomination $q \mapsto [\ell_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$ and to return q as its result. After a clone, the names p and q denote two copies of the same denotation $[\ell_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$; updates to one will not affect the other.

A let *let* x=a *in* b is an expression that first runs the expression a, and if it returns a result, calls it x, and then runs the expression b.

A parallel composition $a \Rsh b$ is either an expression or a process, depending on whether b is an expression or a process. In $a \Rsh b$ the terms a and b are running in parallel. If b is an expression then $a \Rsh b$ is an expression, whose

result, if any, is the result returned by b. Any result returned by a is ignored.

A restriction $(\nu p)a$ is either an expression or a process, depending on whether a is an expression or a process. A restriction $(\nu p)a$ generates a fresh name p whose scope is a.

2.3 Formal Semantics

We base our operational semantics on structural congruence and reduction relations. Reduction represents individual computation steps, and is defined in terms of structural congruence. Structural congruence allows the rearrangement of the syntactic structure of a term so that reduction rules may be applied. We may regard our semantics as a concurrent extension of the small-step substitution-based semantics of **impς** described by Gordon, Hankin, and Lassen [11].

The most interesting aspect of our formal semantics is the management of concurrent expressions that return results. We intend that the result of an expression be that returned from the right-hand side of the topmost parallel composition. Therefore, as we discussed in Section 1, in contexts expecting a result, parallel composition is not commutative. On the other hand, in contexts immediately to the left of a parallel composition, where any result is discarded, parallel composition is commutative. Therefore, structural congruence identifies $(a \Rsh b) \Rsh c$ with $(b \Rsh a) \Rsh c$, since any results returned by a or b are discarded.

Let *structural congruence* be the least congruence on terms to satisfy:

Structural congruence $a \equiv b$

$(a \Rsh b) \Rsh c \equiv a \Rsh (b \Rsh c)$	
$(a \Rsh b) \Rsh c \equiv (b \Rsh a) \Rsh c$	
$(\nu p)(\nu q)a \equiv (\nu q)(\nu p)a$	
$(\nu p)(a \Rsh b) \equiv a \Rsh (\nu p)b$	if $p \notin fn(a)$
$(\nu p)(a \Rsh b) \equiv ((\nu p)a) \Rsh b$	if $p \notin fn(b)$
$\mathit{let}\ x{=}(\mathit{let}\ y{=}a\ \mathit{in}\ b)\ \mathit{in}\ c \equiv \mathit{let}\ y{=}a\ \mathit{in}\ (\mathit{let}\ x{=}b\ \mathit{in}\ c)$	if $y \notin fv(c)$
$(\nu p)\mathit{let}\ x{=}a\ \mathit{in}\ b \equiv \mathit{let}\ x{=}(\nu p)a\ \mathit{in}\ b$	if $p \notin fn(b)$
$a \Rsh \mathit{let}\ x{=}b\ \mathit{in}\ c \equiv \mathit{let}\ x{=}(a \Rsh b)\ \mathit{in}\ c$	

Let *reduction* be the least relation on terms to satisfy:

Reduction $a \to b$

For the first three rules, let $d = [\ell_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$.	
$(p \mapsto d) \Rsh p.\ell_j \to (p \mapsto d) \Rsh b_j\{\!\{x_j \leftarrow p\}\!\}$	if $j \in 1..n$
$(p \mapsto d) \Rsh (p.\ell_j \Leftarrow \varsigma(x)b) \to (p \mapsto d') \Rsh p$	if $j \in 1..n$, $d' = [\ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i \ ^{i \in (1..n)-\{j\}}]$
$(p \mapsto d) \Rsh \mathit{clone}(p) \to (p \mapsto d) \Rsh (\nu q)(q \mapsto d \Rsh q)$	if $q \notin fn(d)$
$\mathit{let}\ x{=}p\ \mathit{in}\ b \to b\{\!\{x \leftarrow p\}\!\}$	
$(\nu p)a \to (\nu p)a'$	if $a \to a'$

$a \Rsh b \to a' \Rsh b$	if $a \to a'$
$b \Rsh a \to b \Rsh a'$	if $a \to a'$
$let\ x{=}a\ in\ b \to let\ x{=}a'\ in\ b$	if $a \to a'$
$a \to b$	if $a \equiv a'$, $a' \to b'$, $b' \equiv b$

We can embed all the expressions of the **impς**-calculus in **concς** via the following abbreviations. If a is not a result, let $a.\ell$, $a.\ell \Leftarrow \varsigma(x)b$, and $clone(a)$ be short for $let\ x{=}a\ in\ x.\ell$, $let\ y{=}a\ in\ y.\ell \Leftarrow \varsigma(x)b$, and $let\ x{=}a\ in\ clone(x)$, respectively. In contexts expecting a term, let an object $[\ell_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]$ be short for the term $(\nu p)(p \mapsto [\ell_i = \varsigma(x_i)b_i\ ^{i\in 1..n}] \Rsh p)$ where $p \notin fn([\ell_i = \varsigma(x_i)b_i\ ^{i\in 1..n}])$. We show in Section 4 that the reductions of any term of **impς** embedded in **concς** are deterministic.

2.4 An Example

The following example from Abadi and Cardelli's book illustrates these abbreviations and the reduction rules for eliminating a let and for method select and update:

$$
\begin{aligned}
&[\ell = \varsigma(x)x.\ell \Leftarrow \varsigma(y)x].\ell \\
&= let\ z{=}[\ell = \varsigma(x)x.\ell \Leftarrow \varsigma(y)x]\ in\ z.\ell \\
&= let\ z{=}(\nu p)(p \mapsto [\ell = \varsigma(x)x.\ell \Leftarrow \varsigma(y)x] \Rsh p)\ in\ z.\ell \\
&\equiv (\nu p)(p \mapsto [\ell = \varsigma(x)x.\ell \Leftarrow \varsigma(y)x] \Rsh let\ z{=}p\ in\ z.\ell) \\
&\to (\nu p)(p \mapsto [\ell = \varsigma(x)x.\ell \Leftarrow \varsigma(y)x] \Rsh p.\ell) \\
&\to (\nu p)(p \mapsto [\ell = \varsigma(x)x.\ell \Leftarrow \varsigma(y)x] \Rsh p.\ell \Leftarrow \varsigma(y)p) \\
&\to (\nu p)(p \mapsto [\ell = \varsigma(y)p] \Rsh p)
\end{aligned}
$$

3 Synchronisation

Since **concς** can express atomic reads and writes on a shared memory, we could use a standard shared memory mutual exclusion algorithm for encoding synchronisation mechanisms. We prefer not to for two reasons. First, such an encoding would be anachronistic since mutual exclusion is normally solved using hardware primitives (such as inhibition of interrupts) rather than reads and writes on a shared memory. Second, such an encoding would lead to complicated calculations about the reduction behaviour of higher level synchronisation mechanisms, such as communication channels.

Instead, we prefer to encode such higher level mechanisms in a calculus $\mathbf{conc\varsigma_m}$ obtained by extending the **concς**-calculus with mutexes (binary semaphores). Unlike shared variable mutual exclusion algorithms, mutexes are commonly used in the runtime systems of object-oriented languages and have simple reduction rules. Still, we have defined a compositional translation of $\mathbf{conc\varsigma_m}$ into **concς**, though we omit it here. We use a two process mutual exclusion algorithm [15] to guarantee exclusive access to the objects representing mutexes.

A third approach would be to add synchronisation mechanisms to the primitive operations on objects, as in the calculus of Di Blasio and Fisher [8]. To keep the primitives of our calculus simple, we prefer not to integrate a specific synchronisation construct into the semantics of method select and method update.

3.1 Syntax

We enrich the syntax to include the denotations *locked* and *unlocked*, and to include the terms $acquire(u)$ and $release(v)$. As before, we adopt the convention that if a denotation d is used as a term, it abbreviates the term $(\nu p)(p \mapsto d \Rsh p)$ for $p \notin fn(d)$. Moreover, if a is not a result, let $acquire(a)$ and $release(a)$ be short for $let\ x{=}a\ in\ acquire(x)$ and $let\ x{=}a\ in\ release(x)$, respectively.

3.2 Informal Semantics

A denomination $p \mapsto locked$ or $p \mapsto unlocked$ represents a mutex, denoted by p, whose state is locked or unlocked, respectively. Intuitively, the mutex is a bit stored at memory location p.

A mutex acquisition $acquire(p)$ attempts to lock the mutex denoted by p. If a denomination $p \mapsto unlocked$ is present, the acquisition $acquire(p)$ changes its state to $p \mapsto locked$, and returns p as its result. Otherwise the acquisition blocks.

A mutex release $release(p)$ unconditionally unlocks the mutex denoted by p. If a denomination $p \mapsto d$ is present, for $d \in \{locked, unlocked\}$, the release $release(p)$ sets its state to $p \mapsto unlocked$, and returns p as its result.

3.3 Formal Semantics

We define the structural congruence relation $\equiv$ by exactly the same rules as in Section 2. The reduction relation $\rightarrow$ is defined by the rules in Section 2 together with two new rules for mutex acquisition and release:

Additional reduction rules

$$(p \mapsto unlocked) \Rsh acquire(p) \rightarrow (p \mapsto locked) \Rsh p$$
$$(p \mapsto d) \Rsh release(p) \rightarrow (p \mapsto unlocked) \Rsh p \qquad \text{for } d \in \{locked, unlocked\}$$

3.4 An Example

We can use mutexes to encode standard forms of synchronisation, such as critical regions and synchronised objects in which at most one method may be active at once. Here we focus on one example: the encoding of asynchronous communications channels similar to those in Pict [21]. Such a channel is an object named by p, that either contains a result or is empty, and has two methods *read* and *write*. If the channel p is empty, the operation $p.write(v)$ updates

p so that it contains v, while the operation $p.read$ blocks. If the channel p contains the result v, the operation $p.read$ returns v and updates p so that it is empty, while the operation $p.write(u)$ blocks. Di Blasio and Fisher [8] implement a similar abstraction in their calculus of concurrent objects. We code channel behaviour as follows. As usual, $a;b$ abbreviates $let\ x{=}a\ in\ b$, where $x \notin fv(b)$. We borrow from the **impς**-calculus an encoding of λ-abstractions $\lambda(x)b$ and function applications $b(a)$ using objects.

$$
\begin{array}{l}
newChan \triangleq \\
\quad let\ rd{=}locked\ in\ let\ wr{=}unlocked\ in \\
\quad [reader = \varsigma(s)rd, writer = \varsigma(s)wr, val = \varsigma(s)s.val, \\
\quad\ read = \varsigma(s)acquire(s.reader); let\ x{=}s.val\ in\ (release(s.writer) \Rsh x), \\
\quad\ write = \varsigma(s)\lambda(x) \\
\quad\quad (acquire(s.writer); s.val \Leftarrow \varsigma(s)x; release(s.reader)) \Rsh x)]
\end{array}
$$

This code maintains the invariant that at any time at most one of the locks *reader* and *writer* is unlocked. If *reader* is unlocked, the result in *val* is the contents of the channel. If *writer* is unlocked, the channel is empty.

Given asynchronous channels, we can encode the asynchronous π-calculus: $[\![\overline{x}y]\!] = x.write(y)$, $[\![x(y).P]\!] = let\ y{=}x.read\ in\ [\![P]\!]$, $[\![P \mid Q]\!] = [\![P]\!] \Rsh [\![Q]\!]$, $[\![(new\ x)P]\!] = let\ x{=}newChan\ in\ [\![P]\!]$, and, for $s \notin \{x, y\} \cup fv(P)$, $[\![!x(y).P]\!] = [rep = \varsigma(s)let\ y{=}x.read\ in\ ([\![P]\!] \Rsh s.rep)].rep$. We conjecture that this translation is sound with respect to a suitable notion of observational equivalence. This particular translation is not fully abstract, since the encoding of channels allows an observer to discover the last message sent on a channel.

4 A Structural Characterisation of Reduction

The purpose of this section is to characterise our reduction semantics in terms of a more conventional structural operational semantics. This is desirable for two reasons. First, it increases our confidence in the correctness of our semantics. Second, it provides a convenient way to enumerate all possible reductions of a term. For the sake of brevity, we work just with **concς**; it is easy to extend our treatment to $\mathbf{conc}\varsigma_{\mathbf{m}}$.

Section 4.1 describes the well-formed terms of **concς** using a rudimentary type system that distinguishes expressions (terms expected to return a result) from processes. In Section 4.2, we demonstrate that on well-formed terms our reduction semantics coincides with a structural operational semantics defined using configurations. Finally, in Section 4.3, we identify a single-threaded fragment of **concς** by omitting a single rule from the rudimentary type system. This fragment is deterministic and includes the **impς**-calculus.

4.1 Well-formed Terms

We present a type system for well-formed terms that distinguishes expressions from processes. In this type system, there are only two types *Proc* and *Exp*. They represent processes and expressions respectively. Since we may always ignore the result of an expression, any term of type *Exp* is also a term of type *Proc*. The type system is very liberal and provides only two guarantees about well-formed terms. First, it guarantees that a proper process does not occur in a context expecting an expression. Second, it guarantees that the top-level denominations of free names in a term represent a partial function from names to objects whose domain is preserved by computation steps.

Let the *domain* of a term a, $dom(a)$, be given by: $dom(p \mapsto d) = \{p\}$, $dom(let\ x{=}a\ in\ b) = dom(a)$, $dom(a \Rsh b) = dom(a) \cup dom(b)$, $dom((\nu p)a) = dom(a) - \{p\}$, and $dom(a) = \varnothing$ for any other kind of a.

Let T stand for either *Proc* or *Exp*. The well-formed terms are given by the judgment $a : T$ defined in the following table. We say that a term a is a *process* or an *expression* if and only if $a : Proc$ or $a : Exp$, respectively.

Well-formed terms

(Well Concur) $\dfrac{a : Exp}{a : Proc}$ (Well Result) $\dfrac{}{u : Exp}$ (Well Object) $\dfrac{b_i : Exp \quad dom(b_i) = \varnothing \quad \forall i \in 1..n}{p \mapsto [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}] : Proc}$

(Well Select) $\dfrac{}{u.\ell : Exp}$ (Well Update) $\dfrac{b : Exp \quad dom(b) = \varnothing}{u.\ell \Leftarrow \varsigma(x)b : Exp}$ (Well Clone) $\dfrac{}{clone(u) : Exp}$ (Well Res) $\dfrac{a : T \quad p \in dom(a)}{(\nu p)a : T}$

(Well Let) $\dfrac{a : Exp \quad b : Exp \quad dom(b) = \varnothing}{let\ x{=}a\ in\ b : Exp}$ (Well Par) $\dfrac{a : Proc \quad b : T \quad dom(a) \cap dom(b) = \varnothing}{a \Rsh b : T}$

Lemma 4.1 *Suppose $a : T$. If $a \equiv b$ or $a \to b$ then $b : T$ and $dom(a) = dom(b)$.*

Terms that are not well-formed include $p \mapsto d_1 \Rsh p \mapsto d_2$, $let\ x{=}p \mapsto d\ in\ b$, $(\nu p)p.\ell$, and $p \mapsto [\ell = \varsigma(x)q \mapsto d]$. None of these receives a type.

4.2 A Structural Operational Semantics

A conventional technique for describing the semantics of concurrent languages with state relies on a syntactic category of configurations, which consist of a store paired with a set of runnable threads. To mimic this technique, we

identify sets of terms that represent threads, stores, and configurations.

Let an *elementary thread*, e, be one of the following: a result, a method update or select, or a clone. Let a *thread*, t, be either an elementary thread, or a term *let* $x{=}t'$ *in* b, where t' is a thread. Let a *store*, σ, be a term of the form $p_1 \mapsto d_1 \Rsh \cdots \Rsh p_m \mapsto d_m$. Let a *configuration*, $(\nu\vec{q})\langle\sigma \parallel t_1, \ldots, t_n\rangle$, be an abbreviation for the term $(\nu\vec{q})(\sigma \Rsh t_1 \Rsh \cdots \Rsh t_n)$.

We may transform any term into a configuration as follows:

Normalising terms to configurations

$$
\begin{array}{lll}
\mathcal{N}(e) & \triangleq & \langle \varnothing \parallel e \rangle \\
\mathcal{N}(p \mapsto d) & \triangleq & \langle p \mapsto d \parallel \varnothing \rangle \\
\mathcal{N}(\mathit{let}\ x{=}a\ \mathit{in}\ b) & \triangleq & (\nu\vec{p})\langle \sigma \parallel \rho, \mathit{let}\ x{=}t\ \mathit{in}\ b \rangle \\
 & & \quad \text{where } \mathcal{N}(a) = (\nu\vec{p})\langle \sigma \parallel \rho, t \rangle \text{ and } \{\vec{p}\} \cap \mathit{fn}(b) = \varnothing \\
\mathcal{N}((\nu p)a) & \triangleq & (\nu p)\mathcal{N}(a) \\
\mathcal{N}(a \Rsh b) & \triangleq & (\nu\vec{p})(\nu\vec{q})\langle \sigma, \sigma' \parallel \rho, \rho' \rangle \\
 & & \quad \text{where } \mathcal{N}(a) = (\nu\vec{p})\langle \sigma \parallel \rho \rangle,\ \mathcal{N}(b) = (\nu\vec{q})\langle \sigma' \parallel \rho' \rangle, \text{ and} \\
 & & \quad \{\vec{p}\} \cap (\mathit{fn}(\sigma') \cup \mathit{fn}(\rho')) = \{\vec{q}\} \cap (\mathit{fn}(\sigma) \cup \mathit{fn}(\rho)) = \varnothing
\end{array}
$$

We can show by induction on the derivation of $a : T$, that $a : T$ implies that $\mathcal{N}(a)$ is well defined and in particular that $T = \mathit{Exp}$ implies that $\mathcal{N}(a)$ takes the form $(\nu\vec{p})\langle \sigma \parallel \rho, t \rangle$.

We define the structural operational semantics to be a relation on terms $a \overset{SOS}{\rightarrow} b$. In the definition, the term a is normalised to a configuration before being reduced to the term b, which is always a configuration.

Structural operational semantics

(SOS Select) (where $\{\vec{p}\} \cap \mathit{fn}(\sigma, \rho_1, \rho_2) = \varnothing$)

$$
\frac{\sigma = \sigma_1, p \mapsto [\ell_i = \varsigma(x_i)b_i\ ^{i\in 1..n}], \sigma_2 \quad j \in 1..n \quad \mathcal{N}(b_j\{\!\{x_j \leftarrow p\}\!\}) = (\nu\vec{p})\langle \sigma' \parallel \rho' \rangle}{\langle \sigma \parallel \rho_1, p.\ell_j, \rho_2 \rangle \overset{SOS}{\rightarrow} (\nu\vec{p})\langle \sigma, \sigma' \parallel \rho_1, \rho', \rho_2 \rangle}
$$

(SOS Update)

$$
\frac{d = [\ell_i = \varsigma(x_i)b_i\ ^{i\in 1..n}] \quad d' = [\ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i\ ^{i\in(1..n)-\{j\}}]}{\langle \sigma_1, p \mapsto d, \sigma_2 \parallel \rho_1, p.\ell_j \Leftarrow \varsigma(x)b, \rho_2 \rangle \overset{SOS}{\rightarrow} \langle \sigma_1, p \mapsto d', \sigma_2 \parallel \rho_1, p, \rho_2 \rangle}
$$

(SOS Clone) (where $q \notin \mathit{fn}(\sigma, \rho_1, \rho_2)$)

$$
\frac{d = [\ell_i = \varsigma(x_i)b_i\ ^{i\in 1..n}] \quad \sigma = \sigma_1, p \mapsto d, \sigma_2}{\langle \sigma \parallel \rho_1, \mathit{clone}(p), \rho_2 \rangle \overset{SOS}{\rightarrow} (\nu q)\langle \sigma, q \mapsto d \parallel \rho_1, q, \rho_2 \rangle}
$$

(SOS Let Result) (where $\{\vec{p}\} \cap fn(\sigma, \rho_1, \rho_2) = \varnothing$)

$$\frac{\mathcal{N}(b\{\!\{x \leftarrow p\}\!\}) = (\nu\vec{p})\langle\sigma' \parallel \rho'\rangle}{\langle\sigma \parallel \rho_1, \mathit{let}\ x{=}p\ \mathit{in}\ b, \rho_2\rangle \stackrel{SOS}{\rightarrow} (\nu\vec{p})\langle\sigma, \sigma' \parallel \rho_1, \rho', \rho_2\rangle}$$

(SOS Let) (where $\{\vec{p}\} \cap fn(\rho_1, b, \rho_2) = \varnothing$)

$$\frac{\langle\sigma \parallel t\rangle \stackrel{SOS}{\rightarrow} (\nu\vec{p})\langle\sigma' \parallel \rho', t'\rangle}{\langle\sigma \parallel \rho_1, \mathit{let}\ x{=}t\ \mathit{in}\ b, \rho_2\rangle \stackrel{SOS}{\rightarrow} (\nu\vec{p})\langle\sigma' \parallel \rho_1, \rho', \mathit{let}\ x{=}t'\ \mathit{in}\ b, \rho_2\rangle}$$

(SOS Res)

$$\frac{a \stackrel{SOS}{\rightarrow} (\nu\vec{p})(\sigma \parallel \rho)}{(\nu p)a \stackrel{SOS}{\rightarrow} (\nu p)(\nu\vec{p})(\sigma \parallel \rho)}$$

(SOS Norm)

$$\frac{\mathcal{N}(a) \stackrel{SOS}{\rightarrow} (\nu\vec{p})\langle\sigma \parallel \rho\rangle}{a \stackrel{SOS}{\rightarrow} (\nu\vec{p})\langle\sigma \parallel \rho\rangle}$$

The structural operational semantics coincides with the reduction semantics up to structural congruence. We write $a \stackrel{SOS}{\rightarrow} \equiv b$ to mean there is c such that $a \stackrel{SOS}{\rightarrow} c$ and $c \equiv b$.

Theorem 4.2 *For all a, b : Exp, $a \rightarrow b$ if and only $a \stackrel{SOS}{\rightarrow} \equiv b$.*

Theorem 4.2 suggests a procedure for discovering all possible reductions of an expression: normalise the expression, then see what $\stackrel{SOS}{\rightarrow}$ reductions are derivable. It is not obvious how to use the $\rightarrow$ relation directly to discover all possible reductions of an expression, since they are defined up to structural congruence.

Theorem 4.2 fails to hold for processes that are not expressions. Consider the process $p.\ell \Rsh p \mapsto [\ell = \varsigma(s)s]$. This term has type *Proc* but not *Exp*. It has no reductions, because composition is not commutative. On the other hand, it is normalised to a configuration $\langle p \mapsto [\ell = \varsigma(s)s] \parallel p.\ell\rangle$ and we have $\langle p \mapsto [\ell = \varsigma(s)s] \parallel p.\ell\rangle \stackrel{SOS}{\rightarrow} \langle p \mapsto [\ell = \varsigma(s)s] \parallel p\rangle$.

The difficulty here is that the reduction relation $a \rightarrow b$ does not represent all of the behaviour of processes that are running as subterms to the left of a composition, where composition is commutative. To remedy this situation, we define versions of structural congruence and reduction specialised to processes situated to the left of a composition. Let $a \stackrel{Proc}{\equiv} b$ if and only if there is $p \notin fn(a) \cup fn(b)$ such that $a \Rsh p \equiv b \Rsh p$. Roughly, $\stackrel{Proc}{\equiv}$ is the same as $\equiv$, except that composition is commutative at the top level. Let $a \stackrel{Proc}{\rightarrow} b$ if and only if $a \stackrel{Proc}{\equiv} a'$, $a' \rightarrow b'$, and $b' \stackrel{Proc}{\equiv} b$. (An alternative definition is to specify these relations by a set of inference rules, simultaneously with the definitions of $a \equiv b$ and $a \rightarrow b$.) We can show that $a \Rsh b \stackrel{Proc}{\equiv} b \Rsh a$ and that $p.\ell \Rsh p \mapsto [\ell = \varsigma(s)s] \stackrel{Proc}{\rightarrow} p \Rsh p \mapsto [\ell = \varsigma(s)s]$. Moreover, we have:

Proposition 4.3 *For all $a, b : Proc$, $a \stackrel{Proc}{\rightarrow} b$ if and only if $a \stackrel{SOSProc}{\rightarrow} \equiv b$.*

4.3 A Single-Threaded Fragment

To identify a deterministic fragment of **concς**, let the *single-threaded type system* for **concς** be the judgment $a :^1 T$ defined by the typing rules except for (Well Concur). We can show that if $a :^1 Proc$ then $a \equiv (\nu\vec{p})\langle\sigma \parallel \varnothing\rangle$, and if $a :^1 Exp$ then $a \equiv (\nu\vec{p})\langle\sigma \parallel t\rangle$. Moreover, we have:

Lemma 4.4 *Suppose $a :^1 Exp$. If $a \equiv b$ or $a \rightarrow b$ then $b :^1 Exp$.*

Theorem 4.5 *Suppose $a :^1 Exp$. If $a \rightarrow a'$ and $a \rightarrow a''$ then $a' \equiv a''$.*

We can show that if a represents a term of **impς**, then $a :^1 Exp$. Hence it follows that **impς** is embedded within a deterministic fragment of **concς** that is closed under reduction.

5 A First-Order Type System

The types of our type system consist of the first-order object types of Abadi and Cardelli's $\mathbf{Ob}_{\mathbf{1}<:}$ together with types for processes and expressions. As in Section 4, we work with **concς**; $\mathbf{conc}_{\mathbf{m}}$ needs an additional type for mutexes.

Let a type A be either $Proc$, Exp, or $[\ell_i : A_i{}^{i\in 1..n}]$, where the ℓ_i are distinct, and $A_i \neq Proc$ for each $i \in 1..n$. As in the rudimentary type system, Exp is the type of expressions, and $Proc$ is the type of processes. As in $\mathbf{Ob}_{\mathbf{1}<:}$, $[\ell_i : A_i{}^{i\in 1..n}]$ is the type of objects with methods $\ell_1, \ldots, \ell_n$ that return results of types $A_1, \ldots, A_n$, respectively. We identify object types up to the reordering of their components. The subtype relation $A <: B$ is the least reflexive and transitive relation on types that satisfies $[\ell_i : B_i{}^{i\in 1..n+m}] <: [\ell_i : B_i{}^{i\in 1..n}]$, $[\ell_i : A_i{}^{i\in 1..n}] <: Exp$, and $Exp <: Proc$.

Let an *environment* E be a list $v_1 : A_1, \ldots, v_n : A_n$; we write $E \vdash \diamond$ to mean that the results v_i are distinct. We define the typing judgment $E \vdash a : A$ as follows:

Typing rules

(Val Subsumption)
$$\frac{E \vdash a : A \quad A <: B}{E \vdash a : B}$$

(Val u)
$$\frac{E, u : A, E' \vdash \diamond}{E, u : A, E' \vdash u : A}$$

(Val Select)
$$\frac{E \vdash u : [\ell_i : B_i{}^{i\in 1..n}] \quad j \in 1..n}{E \vdash u.\ell_j : B_j}$$

(Val Object) (where $A = [\ell_i : B_i{}^{i\in 1..n}]$)
$$\frac{E = E_1, p : A, E_2 \quad E, x_i : A \vdash b_i : B_i \quad dom(b_i) = \varnothing \quad \forall i \in 1..n}{E \vdash p \mapsto [\ell_i = \varsigma(x_i)b_i{}^{i\in 1..n}] : Proc}$$

(Val Update) (where $A = [\ell_i : B_i{}^{i\in 1..n}]$)

$$\frac{E \vdash u : A \quad j \in 1..n \quad E, x : A \vdash b : B_j \quad dom(b) = \varnothing}{E \vdash u.\ell_j \Leftarrow \varsigma(x)b : A}$$

(Val Clone)

$$\frac{E \vdash u : [\ell_i : B_i{}^{i\in 1..n}]}{E \vdash clone(u) : [\ell_i : B_i{}^{i\in 1..n}]}$$

(Val Let) (where $A <: Exp$ and $B <: Exp$)

$$\frac{E \vdash a : A \quad E, x : A \vdash b : B \quad dom(b) = \varnothing}{E \vdash let\ x{=}a\ in\ b : B}$$

(Val Par) (where $dom(a) \cap dom(b) = \varnothing$)

$$\frac{E \vdash a : Proc \quad E \vdash b : B}{E \vdash a \Rsh b : B}$$

(Val Res)

$$\frac{E, p : A \vdash a : B \quad p \in dom(a)}{E \vdash (\nu p)a : B}$$

These rules are a straightforward combination of the rules of Abadi and Cardelli's $\mathbf{Ob_{1<:}}$ and the rules of the rudimentary type system from Section 4.

Lemma 5.1 *If $E \vdash a : A$, $A <: T$, and $T \in \{Proc, Exp\}$ then $a : T$.*

Theorem 5.2 *Suppose $E \vdash a : A$. If $a \equiv b$ or $a \to b$ then $E \vdash b : A$.*

To prove a subject reduction theorem like Theorem 5.2 for typed forms of **impς**, Abadi and Cardelli need to introduce the standard auxiliary notion of store typing. Since the terms of our calculus include both sequential threads and stores, we have no need to separate the notion of store typing from the notion of a typable term. The outcome is a crisper statement of subject reduction than for the imperative form of $\mathbf{Ob_{1<:}}$ in Abadi and Cardelli's book.

The forms of structural congruence and reduction specialised to processes situated to the left of a composition preserve typing at type *Proc*:

Proposition 5.3 *Suppose $E \vdash a : Proc$. If $a \overset{Proc}{\equiv} b$ or $a \overset{Proc}{\to} b$ then $E \vdash b : Proc$.*

Let $A \to B$ be short for $[arg : A, val : B]$, as usual in object calculi. Let $\updownarrow A$ be the type $[read : A, write : A \to A]$. Using subsumption to hide the internal methods *reader*, *writer*, and *val*, we get $\varnothing \vdash newChan : \updownarrow A$ where $\varnothing$ is the empty environment. To further refine usage of these channel types we define a type of write-only channels, $\uparrow A = [write : A \to A]$, and a type of read-only channels, $\downarrow A = [read : A]$, as in the work of Pierce and Sangiorgi [19]. The inclusions $\updownarrow A <: \uparrow A$ and $\updownarrow A <: \downarrow A$ are part of the definition of Pierce and Sangiorgi's system but are derivable in ours.

6 Conclusions

We described a concurrent extension of Abadi and Cardelli's imperative object calculus, **impς**. The syntax of our calculus is essentially that of **impς** together with parallel composition and restriction from the π-calculus, and new primitives for synchronisation via mutexes. This syntax is extremely expressive; in a precise sense it unifies notions of expression, process, store, thread, and configuration. We presented a novel reduction semantics for concurrent expressions, without any need for evaluation contexts, and proved that it corresponds to a more conventional structural operational semantics defined in terms of configurations. We exhibited translations of the asynchronous π-calculus and the **impς**-calculus into our calculus, and showed that it supports the first-order type system $\mathbf{Ob}_{\mathbf{1}<:}$ of objects with subtyping.

Our translations of π and **impς** into our calculus raise questions concerning observational equivalences that we intend to study in future work. Another avenue to investigate is the encoding of other concurrency primitives, like monitors, condition variables, and named threads.

Acknowledgements

Thanks to Alan Jeffrey and Søren Lassen for useful conversations about concurrent objects. Martín Abadi, Luca Cardelli, Søren Lassen, and Andy Pitts commented on a draft of this paper. This work was supported by a Royal Society University Research Fellowship, an EPSRC Research Studentship, and by the EPSRC project "An Operational Theory of Objects". Gordon's current affiliation is Microsoft Research.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1), January 1997.

[3] R. Amadio, L. Leth, and B. Thomsen. From a concurrent λ-calculus to the π-calculus. In *Proceedings Foundations of Computation Theory 95*, volume 965 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[4] D. Berry, R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *Proceedings POPL'92*, pages 119–129, 1992.

[5] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

[6] G. Boudol. The pi-calculus in direct style. In *Proceedings POPL'97*, pages 228–241, 1997.

[7] S. Dal Zilio. Concurrent objects in the blue calculus, 1998. Draft.

[8] P. Di Blasio and K. Fisher. A calculus for concurrent objects. In *Proceedings CONCUR'96*, August 1996.

[9] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisimulation for core CML. Technical Report 95:05, Computer Science, School of Cognitive and Computing Sciences, University of Sussex, 1995.

[10] C. Fournet and G. Gonthier. The reflexive CHAM and the Join-calculus. In *Proceedings POPL'96*, pages 372–385, January 1996.

[11] A. D. Gordon, P. D. Hankin, and S. B. Lassen. Compilation and equivalence of imperative objects. In *Proceedings FST&TCS'97*, volume 1346 of *Lecture Notes in Computer Science*, pages 74–87. Springer-Verlag, 1997. Full version available as Technical Report 429, University of Cambridge Computer Laboratory, 1997.

[12] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.

[13] C. Jones. A pi-calculus semantics for an object-based design notation. In *Proceedings CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 1993.

[14] J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. In *Proceedings PROCOMET'98*, 1998.

[15] L. Lamport. A fast mutual exclusion algorithm. Technical Report 7, Digital Systems Research Center, November 1985.

[16] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.

[17] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–77, 1992.

[18] S. L. Peyton Jones, A. D. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings POPL'96*, pages 295–308, 1996.

[19] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. Summary in *Proceedings LICS'93*, pp. 376–385 (1993).

[20] B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In *Proceedings TPPP'94*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215. Springer-Verlag, 1995.

[21] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, editors, MIT Press, 1998.

[22] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN–19, DAIMI, Aarhus University, September 1981.

[23] J. H. Reppy. *Higher-Order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, 1992. Available as Technical Report 92–1285.

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

[25] D. Walker. Objects in the pi-calculus. *Information and Computation*, 116(2):253–271, February 1995.