Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore*

Stephan Brandauer[1], Elias Castegren[1], Dave Clarke[1], Kiko Fernandez-Reyes[1], Einar Broch Johnsen[2], Ka I Pun[2], S. Lizeth Tapia Tarifa[2], Tobias Wrigstad[1], and Albert Mingkun Yang[1]

[1] Dept. Information Technology, Uppsala University, Sweden
[2] Department of Informatics, University of Oslo, Norway

Abstract The age of multi-core computers is upon us, yet current programming languages, typically designed for single-core computers and adapted *post hoc* for multi-cores, remain tied to the constraints of a sequential mindset and are thus in many ways inadequate. New programming language designs are required that break away from this old-fashioned mindset. To address this need, we have been developing a new programming language called Encore, in the context of the European Project UpScale. The paper presents a motivation for the Encore language, examples of its main constructs, several larger programs, a formalisation of its core, and a discussion of some future directions our work will take. The work is ongoing and we started more or less from scratch. That means that a lot of work has to be done, but also that we need not be tied to decisions made for sequential language designs. Any design decision can be made in favour of good performance and scalability. For this reason, Encore offers an interesting platform for future exploration into object-oriented parallel programming.

1 Introduction

Nowadays the most feasible way for hardware manufacturers to produce processors with higher performance is by putting more parallel cores onto a single chip. This means that virtually every computer produced these days is a parallel computer. This trend is only going to continue: machines sitting on our desks are already parallel computers, and massively parallel computers will soon be readily at our disposal.

Most current programming languages were defined to be sequential-by-default and do not always address the needs of the multi-core era. Writing parallel programs in these languages is often difficult and error prone due to race conditions and the challenges of exploiting the memory hierarchy effectively. But because every computer will be a parallel computer, every programmer needs to become

* Partly funded by the EU project FP7-612985 UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimisations (`http://www.upscale-project.eu`).

a parallel programmer supported by general-purpose parallel programming languages. A major challenge in achieving this is supporting scalability, that is, allowing execution times to remain stable as both the size of the data and available parallel cores increases, without obfuscating the code with arbitrarily complex synchronisation or memory layout directives.

To address this need, we have been developing the parallel programming language Encore in the context of the European Project UpScale. The project has one ambitious goal: to develop a general purpose parallel programming language (in the object-oriented vein) that supports scalable performance. Because message-based concurrency is inherently more scalable, UpScale takes actor-based concurrency, asynchronous communication, and guaranteed race freedom as the starting points in the development of Encore.

Encore is based on (at least) four key ingredients: *active object parallelism* for coarse-grained parallelism, *unshared local heaps* to avoid race conditions and promote locality, *capabilities for concurrency control* to enable safe sharing, and *parallel combinators* for expressing high-level coordination of active objects and low-level data parallelism. The model of active object parallelism is based on that of languages such as Creol [22] and ABS [21]. It requires sequentialised execution inside each active object, but parallel execution of different active objects in the system. The core of the local heaps model is a careful treatment of references to passive objects so that they remain within an active object boundary. This is based on Joëlle [14] and involves so-called sheep cloning [11,30] to copy arguments passed to methods of other active objects. Sheep cloning is a variant of deep cloning that does not clone references to futures and active objects. Capabilities allow these restrictions to be lifted in various ways to help unhinge internal parallelism while still guaranteeing race free execution. This is done using type-based machinery to ensure safe sharing, namely that no unsynchronised mutable object is shared between two different active objects. Finally, Encore includes parallel combinators, which are higher-order coordination primitives, derived from Orc [23] and Haskell [31], that sit both on top of objects providing high-level coordination and within objects providing low-level data parallelism.

This work describes the Encore language in a tutorial fashion, covering course-grained parallel computations expressible using active objects, and fine-grained computations expressible using higher-order functions and parallel combinators. We describe how these integrate together in a safe fashion using capabilities and present a formalism for a core fragment of Encore.

Currently, the work on Encore is ongoing and our compiler already achieves good performance on some benchmarks. Development started more or less from scratch, which means not only that we have to build a lot of infrastructure, but also that we are free to experiment with different implementation possibilities and choose the best one. We can modify anything in the software stack, such as the memory allocation strategy, and information collected about the program in higher levels can readily be carried from to lower levels—contrast this with languages compiled to the Java VM: source level information is effectively lost in

translation and VMs typically do not offer much support in controlling memory layout, etc.

ENCORE has only been under development for about a year and a half, consequently, anything in the language design and its implementation can change. This tutorial therefore can only give a snapshot of what ENCORE aspires to be.

Structure. Section 2 covers the active object programming model. Section 3 gently introduces ENCORE. Section 4 discusses active and passive classes in ENCORE. Section 5 details the different kinds of methods available. Section 6 describes futures, one of the key constructs for coordinating active objects. Section 7 enumerates many of the commonplace features of ENCORE. Section 8 presents a stream abstraction. Section 9 proposes parallel combinators as a way of expressing bulk parallel operations. Section 10 advances a particular capability system as a way of avoiding data races. Section 11 illustrates ENCORE in use via examples. Section 12 formalises a core of ENCORE. Section 13 explores some related work. Finally, Section 14 concludes.

2 Background: Active Object-based Parallelism

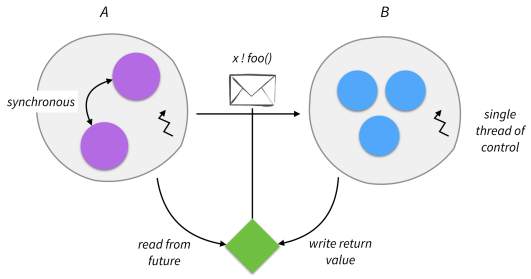


Figure 1. Active Object-based Parallelism

ENCORE is an active object-based parallel programming language. Active objects (Figure 1), and their close relation, actors, are similar to regular object-oriented objects in that they have a collection of encapsulated fields and methods that operate on those fields, but the concurrency model is quite different from what is found in, for example, Java [17]. Instead of threads trampling over all objects, hampered only by the occasional lock, the active-object model associates a thread of control with each active object, and this thread is the only one able to access the active object's fields. Active objects communicate with each other by sending messages (essentially method calls). The messages are placed in a queue associated with the target of the message. The target active object processes the

messages in the queue one at a time. Thus at most one method invocation is active at a time within an active object.

Method calls between active objects are asynchronous. This means that when an active object calls a method on another active object, the method call returns immediately—though the method does not run immediately. The result of the method call is a future, which is a holder for the eventual result of the method call. The caller can do other work immediately, and when it needs the result of the method call, it can *get* the value from the future. If the value is not yet available, the caller blocks.

Futures can be passed around, blocked on (in various ways), or have additional functionality chained on them. This last feature, available in Javascript for instance, allows the programmer to chain multiple asynchronous computations together in a way that makes the program easy to understand by avoiding callbacks.

Actors are a similar model to active objects (though often the terminology for describing them differs). Two features are more commonly associated with active objects. Firstly, active objects are constructed from (active) classes, which typically are composed using inheritance and other well-known object-oriented techniques. This arguably makes active objects easier to program with as they are closer to what many programmers are used to. Secondly, message sends (method calls) in active objects generally return a result, via a future, whereas message sends in actors are one-way and results are obtained via a callback. Futures are thus key to making asynchronous calls appear synchronous and avoid the inversion of control associated with callbacks.

Weakness of Active Objects Although active objects have been selected as the core means for expressing parallel computation in Encore, the model is not without limitations. Indeed, much of our research will focus on ways of overcoming these.

Although futures alleviate the problem of inversion of control described above in problem, they are not without code. Waiting on a future that had not been fulfilled can be expensive as it involves blocking the active object's thread of control, which may then prevent other calls depending in the active object to block. Indeed, the current implementation of blocking on a future in Encore is costly.

A second weakness of active objects is that, at least in the original model, it is impossible to execute multiple concurrent method invocations within an active object, even if these method invocations would not interfere. Some solutions to this problem have been proposed [20] allowing a handful of method invocations to run in parallel, but these approaches do not unleash vast amounts of parallelism and they lack any means for structuring and composing the non-interfering method invocations. For scalability, something more is required. Our first ideas in this direction are presented in Section 9.

3 Hello Encore

ENCORE programs are stored in files with the suffix `.enc` by convention and are compiled to C. The generated C code is compiled and linked with the ENCORE run-time system, which is also written in C. The compiler itself is written in Haskell, and the generated C is quite readable, which significantly helps with debugging.

To get a feeling for how ENCORE programs look, consider the following simple program (in file `hello.enc`) that prints "Hello, World!" to standard output.

```
1 #! /usr/bin/env encorec -run
2 class Main
3   def main() : void {
4     print("Hello, World!")
5   }
```

The code is quite similar to modern object-oriented programming languages such as Scala, Python or Ruby. It is statically typed, though many type annotations can be omitted, and, in many cases, the curly braces `{` and `}` around classes, method bodies, etc. can also be omitted.

Ignoring the first line for now, this file defines an active class `Main` that has a single method `main` that specifies its return type as **`void`**. The body of the method calls `print` on the string `"Hello, World!"`, and the behaviour is as expected.

Every legal ENCORE program must have an active class called `Main`, with a method called `main`—this is the entry point to an ENCORE program. The run-time allocates one object of class `Main` and begins execution in its `main` method.

The first line of `hello.enc` is optional and allows the compiler to automatically compile and run the program (on Unix systems such as Mac OS X). The file `hello.enc` has to be runnable, which is done by executing `chmod u+x hello.enc` in the shell. After making the program executable, entering `./hello.enc` in the shell compiles and executes the generated binary, as follows:

```
$ ./hello.enc
Hello, World!
```

An alternative to the `#! /usr/bin/env encorec -run` line is to call the compiler directly, and then run the executable:

```
$ encorec hello.enc
$ ./hello
Hello, World!
```

4 Classes

ENCORE offers both active and passive classes. Instances of active classes, that is, *active objects*, have their own thread of control and message queue (cf.

Section 2). Making all objects active would surely consume too many system resources and make programming difficult, so passive objects are also included in ENCORE. *Passive objects*, instances of passive classes, do not have a thread of control. Passive classes are thus analogous to (unsynchronised) classes in mainstream object-oriented languages like Java or Scala. Classes are active by default: `class A`. The keyword **`passive`** added to a class declaration makes the class passive: **`passive class`** `P`. Valid class names must start with an uppercase letter. (Type parameters start with a lower case letter.) Classes in ENCORE have fields and methods; there is a planned extension to include traits and interfaces integrating with capabilities (cf. Section 10).

A method call on an active object will result in a message being placed in the active object's message queue and the method invocation possibly runs in parallel with the callee. The method call immediately results in a future, which will hold the eventual result of the invocation (cf. Section 6). A method call on a passive object will be executed synchronously by the calling thread of control.

4.1 Object Construction and Constructors

Objects are created from classes using **`new`**, the class name and an optional parameter list: **`new`** `Foo(42)`. The parameter list is required if the class has an `init` method, which is used as the constructor. This constructor method cannot be called on its own in other situations.

4.2 Active Classes

The following example illustrates active classes. It consists of a class `Buffer` that wraps a `Queue` data structure constructed using passive objects (omitted). The active object provides concurrency control to protect the invariants of the underlying queue, enabling the data structure to be shared. (In this particular implementation, taking an element from the `Buffer` is implemented using `suspend` semantics, which is introduced in Section 7.)

```
1  passive class Data { ... }
2  class Buffer
3    queue : Queue;
4
5    def init()
6      this.queue = new Queue()
7
8    def put(item : Data) : void
9      this.queue.enqueue(item)
10
11   def take() : Data {
12     while this.queue.empty() {
13       suspend;
14     };
15     this.queue.dequeue()
16   }
```

Fields of an active object are private; they can only be accessed via **this**, so the field queue of Buffer is inaccessible to an object holding a reference to a Buffer object.

4.3 Passive Classes

Passive classes in ENCORE correspond to regular (unsynchronised) classes in other languages. Passive classes are used for representing the state of active objects and data passed between active objects. Passive classes are indicated with the keyword **passive**.

```
1 passive class Person {
2   name : string
3   age : int
4
5   def init(name : string, age : int) {
6     this.name = name;
7     this.age = age;
8   }
9 }
```

In passive classes, all fields are public:

```
1 class Main
2   def main() : void
3     let p = new Person("Dave", 21) in
4       print("Hello {}\n", p.name) -- prints "Hello Dave"
```

4.4 Parametric Classes

Classes can take type parameters. This allows, for example, parameterised pairs to be implemented:

```
1 passive class Pair<a, b>
2   fst : a
3   snd : b
4   def init(fst_ : a, snd_ : b) : void {
5     this.fst = fst_;
6     this.snd = snd_
7   }
```

This class can be used as follows:

```
1 class Main
2   def main() : void
3     let pair = new Pair<int,string>(65, "a") in
4       print("({},{})\n", pair.fst, pair.snd)
```

Currently, type parameters are unbounded in ENCORE, but this limitation will be removed in the future.

4.5 Traits and Inheritance

ENCORE is being extended with support for *traits* [15] to be used in place of standard class-based inheritance. A trait is a composable unit of behaviour that

provides a set of methods and requires a set of fields and methods from any class that wishes to include it. The exact nature of ENCORE traits has not yet been decided at time of writing.

A class may be self-contained, which is the case for classes shown so far and most classes shown in the remainder of this document, or be constructed from a set of pre-existing traits. The inclusion order of traits is insignificant, and multiple ways to combine traits are used by the type system to reason about data races (cf. Section 10). Below, the trait `Comparator` implementation requires that the including class defines a `cmp` method, and provides five more high-level methods all relying on the required method.

```
trait Comparator<t>
  require def cmp(t): int;

  def equal(v:t) : bool
    this.cmp(v) == 0

  def lessThan(v:t) : bool
    this.cmp(v) < 0
  def lessThanOrEqual(v:t) : bool
    this.cmp(v) <= 0

  def greaterThan(v:t) : bool
    this.cmp(v) > 0
  def greaterThanOrEqual(v:t) : bool
    this.cmp(v) >= 0
```

Traits enable trait-based polymorphism—it is possible, for instance, to write a method that operates on any object whose class includes the `Comparator` trait:

```
def contains(p:person, ps:[Comparator<Person>]) : bool
  let
    found = false
    size = |ps|
    i = 0
  in {
    while not found and i < size
      {
        if ps[i].equal(p) then found = true;
        i = i + 1;
      }
    return found;
  }
```

For more examples of traits, see Section 10.

5 Method Calls

Method calls may run asynchronously (returning a future) or synchronously depending primarily on whether the target is active or passive. The complete range of possibilities is given in the following table:

	Synchronous	Asynchronous
Active objects	`get o.m()`	`o.m()`
Passive objects	`o.m()`	—
`this` (in Active)	`this.m()`	`let that = this in that.m()`

Self calls on active objects can be run synchronously—the method called is run immediately—or asynchronously—a future is immediately returned and the invocation is placed in the active object's queue.

Sometimes the result of an asynchronous method call is not required, and savings in time and resources can be gained by not creating the data structure implementing the future. To inform the compiler of this choice, the `.` in the method call syntax is replaced by a `!`, as in the following snippet:

```
1 cart ! add_item(item)
```

6 Futures

Method calls on active objects run asynchronously, meaning that the method call is run potentially by a different active object and that the current active object does not wait for the result. Instead of returning a result of the expected type, the method call returns an object called a *future*. If the return type of the method is `t`, then a value of type `Fut t` is returned to the caller. A future of type `Fut t` is a container that at some point *in the future* will hold a value of type `t`, typically when some asynchronous computation finishes. When the asynchronous method call finishes, it writes its result to the future, which is said to be *fulfilled*. Futures are considered first class citizens, and can be passed to and returned from methods, and stored in data types. Holding a value of type `Fut t` gives a hint that there is some parallel computation going on to fulfil this future. This view of a future as a handle to a parallel computation is exploited further in Section 9.

Several primitive operations are available on futures:

- `get : Fut t -> t` waits for the future to be fulfilled, blocking the current active object until it is; returns the value stored in the future.
- `await : Fut t -> void` waits for the future to be fulfilled, without blocking the current active object, thus other methods can run; does not return a value.[3]
- chaining: `~~> : Fut t -> (t -> t') -> Fut t'` takes a closure to run on the result when the future is fulfilled; returns another future that will contain the result of running the closure.

These operations will be illustrated using following the classes as a basis. These classes model a service provider that produces a certain product:

[3] This design should change, so that `await` will become more similar to `get`, but with a different effect on the active object.

```
1 passive class Product { ... }
2 class Service {
3   def provide() : Product {
4     new Product()
5   }
```

The next subsections provide several implementations of clients that call on the service provider, create an instance of class `Handle` to deal with the result, and pass the result provided by the service provider to the handler.

6.1 Using the `get` operation

When the **`get`** operation is applied to a future, the current active object blocks until the future is fulfilled, and when it has been, the call to **`get`** returns the value stored in the future.

Consider the following client code.

```
1  class Handler { ... }
2  class Client
3    service : Service
4
5    def run() : void {
6      let fut = service.provide()
7          handler = new Handler()
8      in {
9        handler.handle(get fut);
10       ...
11     }
12   }
```

In method `run` of `Client`, the call to `service.provide()` results in a future of type **`Fut`** `Product` (line 6). In line 9, the actual `Product` object is obtained using a call to **`get`**. If the future had already been fulfilled, the `Product` object would be returned immediately. If not, method *and* the active object block, preventing any progress locally until the future is fulfilled.

6.2 Using the `await` command

One of the problems with calling **`get`** on a future is that it can result in the entire active object being blocked—sometimes this is desirable to ensure than internal invariants hold, but it can result in costly delays, for example, if the method called involves a time-consuming calculation. During that time, the whole active object can make no progress, which would also block other active objects that need its services.

An alternative, when it makes sense, is to allow the active object to process other messages from its message queue and resume the current method call sometime after the future has been fulfilled. This is exactly what calling **`await`** on a future allows.

Command **`await`** applies to a future and waits for it to be fulfilled, blocking the current method call but without blocking the current active object. The call

to **await** does not return a value, so a call to **get** is required to get the value. This call to **get** is guaranteed to succeed without blocking.

The following code provides an alternative implementation to the method run from class `Client` above using **await**:[4]

```
1 def run() : void {
2   let fut = service.provide()
3       handler = new Handler()
4   in {
5     await fut;
6     handler.handle(get fut);
7     ...
8   }
9 }
```

In this code, the call **await** `fut` on line 5 will block if the future `fut` is unfulfilled; other methods could run in the same active object between lines 5 and 6. When control returns to this method invocation, execution will resume on line 6 and the call to **get** `fut` is guaranteed to succeed.

6.3 Using future chaining

The final operation of futures is future chaining (`fut ~~> g`) [25]. Instead of waiting for the future `fut` to be fulfilled, as is the case for **get** and **await**, future chaining attaches a closure `g` to the future to run when the future is fulfilled. Future chaining immediately returns a future that will store the result of applying the closure to the result of the original future.

The terminology comes from the fact that one can add a further closure onto the future returned by future chaining, and add a additional closure onto that, and so forth, creating a chain of computations to run asynchronously. If the code is written in a suitably stylised way (e.g., one of the ways of writing monadic code such as Haskell's do-notation [31]), then the code reads in sequential order—no inversion of control

Consider the following alternative implementation of the `run` method from class `Client` above using future chaining:

```
1 def run() : void {
2   let fut = service.provide()
3       handler = new Handler()
4   in {
5     fut ~~> (\(prod: Producer) -> handler.handle(prod)) -- future chaining
6     ...
7   }
8 }
```

In the example above, the closure defined on line 5 will be executed as soon as the future from `service.provide()` (line 2) is fulfilled.

A chained closure can run in one of two modes, depending on what is accessed within the closure. If the closure accesses fields or passive objects from the

[4] Ideally, this should be: `handler.handle(await fut)`. Future versions of Encore will support this semantics.

surrounding context, which would create the possibility of race conditions, then it must be run in *attached* mode, meaning that the closure when invoked will be run by the active object that lexically encloses it. The closure in the example above needs to run in attached mode as it accesses the local variable `handle`. In contrast, a closure that cannot cause race conditions with the surrounding active object can be run in *detached* mode, which means that it can be run independently of the active object. To support the specification of detached closures, the notion of spore [27], which is a closure with a pre-specified environment, can be used (cf. Section 7.6). Capabilities (Section 10) will also provide means for allowing safe detached closures.

7 Expressions, statements, and so forth

ENCORE has many of the language features one expects from a general purpose programming language. Some of these features are described (briefly) in the following subsections.

7.1 Types

ENCORE has a number of built in types. The following table presents these, along with typical literals for each type:

Type	Description	Literals
`void`	the unit value	`()`
`string`	strings	`"hello"`
`int`	fixed-precision integers	`1, -12`
`uint`	unsigned, fixed-precision integers	`42`
`real`	floating point numbers	`1.234, -3.141592`
`bool`	booleans	**`true`**, **`false`**
`Fut` `t`	futures of type `t`	—
`Par` `t`	parallel computations producing type `t`	—
`Stream` `t`	functional streams of type `t`	—
`t -> t'`	functions from type `t` to type `t'`	`\x -> x * 2`
`[t]`	arrays of type `t`	`[1,2,3,6]`, but not `[]`

The programmer can also introduce two new kinds of types: active class types and passive classes types, both of which can be polymorphic (cf. Section 4).

7.2 Expression Sequences

Syntactically, method bodies, while bodies, let bodies, etc. consist of a single expression:

```
1 def single() : void
2   print "a single expression needs no curly braces"
```

In this case, curly braces are optional.

```
def curly() : void {
  print ".. but it CAN use them!"
}
```

If several expressions need to be sequenced together, this is done by separating them by semicolons and wrapping them in curly braces. The value of a sequence is the value of its last expression. A sequence can be used wherever an expression is expected.

```
def multiple() : int {
    print "multiple";
    print "expressions";
    print "are wrapped by { ... }";
    print "and separated by ';'";
    2
}
```

7.3 Loops

ENCORE has two kinds of loops: **while** and **repeat** loops. A **while** loop takes a boolean loop condition, and evaluates its body expression repeatedly, as long as the loop condition evaluates to true:

```
let i = 0 in
  while i < 5 {
    print("i={}\n",i);
    i = i + 1
  }
```

This prints:

```
i=0
i=1
i=2
i=3
i=4
```

The **repeat** loop is syntax sugar that makes iterating over integers simpler. The following example is equivalent to the **while** loop above:

```
repeat i <- 5
  print("i={}\n",i)
```

In general,

```
repeat i <- n
  expr
```

evaluates `expr` for values $i = 0, 1, \ldots, n - 1$.

7.4 Arrays

The type of arrays of type `T` is denoted `[T]`. An array of length `n` is created using **new** `[T](n)`. Arrays are indexed starting from 0. Arrays are fixed in size and cannot be dynamically extended or shrunk.

Array elements are accessed using the bracket notation: `a[i]` accesses the ith element. The length of an array is given by `|a|`. Arrays can be constructed using the literal notation `[1, 2, 1+2]`.

The following example illustrates the features of arrays:

```
class Main
  def bump(arr: [int]): void
      repeat i <- |arr|
        arr[i] = arr[i] + 1

  def main(): void {
    let a = [1,2,3] in {
      this.bump(a);
      repeat i <- |a|
        print a[i];
      let b = new [int](3) in {
        b[0] = 0;
        b[1] = a[0];
        b[2] = 42 - 19;
      };
      repeat i <- |b|
        print b[i];
    }
  }
```

The expected output is

```
2
3
4
0
2
23
```

7.5 Formatted printing

The `print` statement allows formatted output. It accepts a variable number of parameters. The first parameter is a format string, which has a number of holes marked with `{}` into which the values of the subsequent parameters are inserted. The number of occurrences of `{}` must match the number of additional parameters.

The following example illustrates how it works.

```
class Main
  def main() : void {
    let i = 0 in {
      while i < 5 {
        i = i+1;
        print("{} * {} = {}\n", i, i, i*i);
      }
    }
  }
```

The output is:

```
$ ./ex_printing.enc
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
```

7.6 Anonymous Functions

In ENCORE, an anonymous function is written as follows:

```
1 \(i : int) -> 10 * i
```

This function multiplies its input i by 10.

The backslash \ (syntax borrowed from Haskell, resembling a lambda) is followed by a comma separated list of parameter declarations, an arrow `->` and an expression, the function body. The return type does not need to be declared as it is always inferred from the body of the lambda.

In the example below, the anonymous function is assigned to the variable `tentimes` and then later applied—it could also be applied directly.

```
1 let tentimes = \(i : int) -> 10 * i in
2       print(tentimes(10)) -- prints 100
```

Anonymous functions are first-class citizens and can be passed as arguments, assigned to variables and returned from methods/functions. Types of functions are declared by specifying its arguments types, an arrow `->`, and the return type. For example, the type of the function above is `int -> int`. Multi-argument functions have types such as `(int, string) -> bool`.

The following example shows how to write a higher-order function `update` that takes a function `f` of type `int -> int`, an array of `int`'s and applies the function `f` to the elements of the array `data`, updating the array in-place.

```
1 def update(f: int -> int, data: [int]): void {
2   repeat i <- |data|
3       data[i] = f(data[i]);
4 }
5
6 class Main
7   def main(): void {
8     let xs = [2,3,4,1] in
9       update(\(data: int) -> data + 1, xs)
10  }
```

Closures as specified above can capture variables appearing in their surrounding lexical context. If a closure is run outside of the context in which it is defined, then data races can occur. A variation on closures exists that helps avoid this problem.

ENCORE provides a special kind of anonymous function called a *spore* [27]. A spore must explicitly specify the elements from its surrounding context that

are captured in the spore. The captured elements can then, more explicitly, be controlled using types, locks or cloning to ensure that the resulting closure can be run outside of the context in which the spore is defined. Spores have an environment section binding the free variables of the sport to values from the surrounding context, and a closure, which can access only those free variables and its parameters.

```
class Provider
  service: Service

  def provide(): Data -> Product {
    spore {
      let x = clone this.service in -- set up environment for closure
      \(y: Data) -> x.produce(y) -- the closure
    }
  }
```

In this code snippet, the only variables in scope in the closure body are `x` and `y`. The field `service`, which would normally be visible within the closure (in Scala or in Java if it were final), is *not* accessible. It is made accessible (actually, a clone of its contents), via variable `x` in the environment section of the spore.

7.7 Polymorphism and Type Inference

At the time of writing, ENCORE offers some support for polymorphic classes, methods and functions. Polymorphism in ENCORE syntactically resembles other well-established OOP languages, such as Java. Type variables in polymorphic classes, methods and/or functions must be written using lower case.

The following example shows how to write a polymorphic list:

```
passive class List<t>
  data: t
  next: List<t>

  def init(data: t): void
    this.data = data

  def append(data: t): void {
    let next_item = new List<t>(this.data) in {
      next_item.next = this.next;
      this.data = data;
      this.next = next_item;
    }
  }
  -- other methods

class Main
  def main(): void {
    let l = new List<int> in {
      l.append(3);
      l.append(4);
    }
  }
```

7.8 Module System

Currently, ENCORE supports a rudimentary module system. The keyword **import** followed by the name of a module imports the corresponding module. The name of the module must match the name of the file, excluding the `.enc` suffix. The compiler looks for the corresponding module in the current directory plus any directories specified using the `-I pathlist` compiler flag.

Assume that a library module `Lib.enc` contains the following code:

```
class Foo
  def boo(): void {
    print "^-^"
  }
```

This module can be imported using **import** `Lib` as illustrated in the following (file `Bar.enc`):

```
import Lib

class Main
  def main(): void {
    let
      f = new Foo
    in
      f.boo()
  }
```

Here `Bar.enc` imports module `Lib` and can thus access the class `Foo`.

Currently, the module system has no notion of namespaces, so all imported objects needs to have unique names. There is also no support for cyclic imports and qualified imports, so it is up to the programmer to ensure that each file is only imported once.

7.9 Suspending execution

The **suspend** command supports cooperative multitasking. It suspends the currently running method invocation on an active object and schedules the invocation to be resumed after all messages in the queue have been processed.

```
class Pi
  def calculate_digits(digits: int): double {
    -- perform initial calculations
    ...
    suspend;
    -- continue performing more calculations
    ...
  }

class Main
  def main(): void {
    let pi = new Pi() in {
      pi.calculate_decimals(100000000000);
    }
  }
```

The example computes a large number of digits of π. The method `calculate_digits` calls **`suspend`** to allow other method calls to run on the `Pi` active object. This is achieved by suspending the execution of the current method call, placing a new message in its message queue, and then releasing control. The message placed in the queue is the continuation of the suspended method invocation, which in this case will resume the suspended method invocation at line 6.

7.10 Embedding of C code

ENCORE supports the embedding of C code. This is useful for wrapping C libraries to import into the generated C code and for experimenting with implementation ideas before incorporating them into the language, code generator, or run-time. Two modes are supported: top-level embed blocks and embedded expressions.

Note that we do not advocate the extensive use of **`embed`**. Code using **`embed`** is quite likely to break with future updates to the language.

Top-level Embed Blocks Each file can contain at most one top-level **`embed`** block, which has to be placed before the first class definition in the file. This **`embed`** block consists of a header section and an implementation section, as in the following example:

```
1 embed
2   int64_t sq(int64_t);
3 body
4   int64_t sq(int64_t n) {
5     return n*n;
6   }
7 end
```

The header section will end up in a header file that all class implementations will include. The implementation section will end up in a separate C file. The `sq` function declaration must be included in the header section, otherwise the definitions in the **`body`** section would not be accessible in the generated C code.

Embedded expressions An **`embed`** block can appear anywhere where an expression can occur. The syntax is:

```
1 embed encore-type C-code end
```

When embedding an expression, the programmer needs to assign an encore type to the expression. ENCORE will assume that this type is correct. The value of an embedded expression is the value of the last C-statement in the embedded code.

ENCORE variables can be accessed from within an **`embed`** block by wrapping them with `#{ }`. For instance, local variable `x` in ENCORE code is accessed using `#{x}` in the embedded C. Accessing fields of the current object is achieved using C's arrow operator. For instance, **`this`**`->foo` accesses the field **`this`**`.foo`.

The following example builds upon the top-level embed block above:

```
class Main
  def main() : void {
    let x = 2 in
      print(embed int sq(#{x}); end)
}
```

The embedded expression in this example promises to return an **int**. It calls the C-function sq on the local Encore variable x.

Embedding C values as abstract data types The following pattern allows C values to be embedded into Encore code and treated as an abstract type, in a sense, where the only operations that can manipulate the C values are implemented in other embedded blocks. In the following code example, a type D is created with no methods or fields. Values of this type cannot be manipulated in Encore code, only passed around and manipulated by the corresponding C code.

```
passive class D

passive class LogArray
  size:int
  slots:D
  def init(size:int) : void
    embed void -- initialise element of type D
    this->slots = pony_alloc(size * sizeof(void*));
    for (int i = 0; i < size; ++i) ((pony_actor_t**)this->slots)[i] = NULL;
    this->size = size;
  end
  def write(i:int, v:LogEntry) : void
    embed void -- modify element of type D
    ((void **)this->slots)[i] = v;
  end
  def read(i:int) : LogEntry
    embed LogEntry --- read element of type D
     ((void **)this->slots)[i];
  end
  def size() : int
    this.size
```

Mapping Encore types to C types The following table documents how Encore's types are mapped to C types. This information is useful when writing embedded C code, though ultimately having some detailed knowledge of how Encore compiles to C will be required to do anything advanced.

ENCORE type	C type
string	(char *)
real	double
int	int64_t
uint	uint64_t
bool	int64_t
⟨an active class type⟩	(encore_actor_t *)
⟨a passive class type⟩	(CLASSNAME_data *)
⟨a type parameter⟩	(**void** *)

8 Streams

A stream in ENCORE is an immutable sequence of values produced asynchronously. Streams are abstract types, but metaphorically, the type **Stream** a can be thought of as the Haskell type:

```
type Stream a = Fut (Maybe (St a))
data St a = St a (Stream a)
```

That is, a stream is essentially a future, because at the time the stream is produced it is unknown what its contents will be. When the next part of contents are known, it will correspond to either the end of the stream (`Nothing` in Haskell) or essentially a pair (`Just (St e s)`) consisting of an element `e` and the rest of the stream `s`.

In ENCORE this metaphor is realised, imperfectly, by making the following operations available for the consumer of a stream:

- **get** : **Stream** a -> a — gets the head element of the (non-empty) stream, blocking if it is not available.
- **getNext** : **Stream** a -> **Stream** a — returns the tail of the (non-empty) stream. A non-destructive operator.
- **eos** : **Stream** a -> Bool — checks whether the stream is empty.

Streams are produced within special **stream** methods. Calling such methods results immediately in a handle to the stream (of type **Stream** a). Within such a method, the command **yield** becomes available to produce values on the stream. **yield** takes a single expression as an argument and places the corresponding value on the stream being produced. When the stream method finishes, stream production finishes and the end of the stream marker is placed in the stream.

The following code illustrate an example stream producer that produces a stream whose elements are of type **int**:

```
class IntSeq
  stream start(fr : int, to : int) : int {
    while fr <= to {
      yield fr;
      fr = fr+1
    };
  }
```

The following code gives an example stream consumer that processes a stream stored in variable `str` of type **`Stream int`**.

```
1  class Main
2    def main() : void
3      let
4        lst = 0
5        str = (new IntSeq).start(1,1000000)
6      in {
7        while not eos str {
8          lst = get str;
9          str = getNext str;
10       };
11       print lst
12     }
```

Notice that the variable `str` is explicitly updated with a reference to the tail of the stream by calling **`getNext`**, as **`getNext`** returns a reference to the tail, rather than updating the object in `str` in place—streams are immutable, not mutable.

9 Parallel Combinators

ENCORE offers preliminary support for parallel types, essentially an abstraction of parallel collections, and parallel combinators that operate on them. The combinators can be used to build pipelines of parallel computations that integrate well with active object-based parallelism.

9.1 Parallel Types

The key ingredient is the parallel type **`Par`** `t`, which can be thought of as a handle to a collection of parallel computations that will eventually produce zero or more values of type `t`—for convenience we will call such an expression a *parallel collection*. (Contrast with parallel collections that are based on a collection of elements of type `t` manipulated using parallel operations [32].) Values of **`Par`** `t` type are first class, thus the handle can be passed around, manipulated and stored in fields of objects.

Parallel types are analogous to future types in a certain sense: an element of type **`Fut`** `t` can be thought of as a handle to a single asynchronous (possibly parallel) computation resulting in a single value of type `t`; similarly, an element of type **`Par`** `t` can be thought of as a handle to a parallel computation resulting in multiple values of type `t`. Pushing the analogy further, **`Par`** `t` can be thought of as a "list" of elements of type **`Fut`** `t`: thus, **`Par`** `t` $\approx$ [**`Fut`** `t`].

Values of type **`Par`** `t` are assumed to be ordered, thus ultimately a sequence of values as in the analogy above, though the order in which the values are produced is unspecified. Key operations on parallel collections typically depend neither on the order the elements appear in the structure nor the order in which they are produced.[5]

[5] An alternative version of **`Par`** `t` is possible where the order in the collection is not preserved. This will be considered in more detail in future experiments.

9.2 A collection of combinators

The operations on parallel types are called parallel combinators. These adapt functionality from programming languages such as Orc [23] and Haskell [31] to express a range of high-level typed coordination patterns, parallel dataflow pipelines, speculative evaluation and pruning, and low-level data parallel computations.

The following are a representative collection of operations on parallel types.[6] Note that all operations are functional.

- `empty : Par t`. A parallel collection with no elements.
- `par : (Par t, Par t) -> Par t`. The expression `par(c, d)` runs `c` and `d` in parallel and results in the values produced by `c` followed (spatially, but not temporally) by the values produced by `d`.
- `pbind : (Par t, t -> Par t') -> Par t'`. The expression `pbind(c,f)` applies the function `f` to all values produced by `c`. The resulting nested parallel collection (of type `Par (Par t')`) is flattened into a single collection (of type `Par t'`), preserving the order among elements at both levels.
- `pmap : (t -> t', Par t) -> Par t'` is a parallel map. The expression `pmap(f,c)` applies the function `f` to each element of parallel collection `c` in parallel resulting in a new parallel collection.
- `filter : (t -> bool, Par t) -> Par t` filters elements. The expression `filter(p, c)` removes from `c` elements that do not satisfy the predicate `p`.
- `select : Par t -> Maybe t` returns the first available result from the parallel type wrapped in tag `Just`, or `Nothing` if it has no results.[7]
- `selectAndKill : Par t -> Maybe t` is similar to `select` except that it also kills all other parallel computations in its argument after the first value has been found.
- `prune : (Fut (Maybe t) -> Par t', Par t) -> Par t'`. The expression `prune(f, c)` creates a future that will hold the result of `selectAndKill(c)` and passes this to `f`. This computation in run in parallel with `c`. The first result of `c` is passed to `f` (via the future), after which `c` is terminated.
- `otherwise : (Par t, () -> Par t) -> Par t`. The expression `otherwise(c, f)` evaluates `c` until it is known whether it will be empty or non-empty. If it is not empty, return `c`, otherwise return `f ()`.

[6] As work in parallel types and combinators is work in progress, this list is likely to change and grow.

[7] Relies on `Maybe` data type: in Haskell syntax `data Maybe a = Nothing | Just a`. Data types are at present being implemented. An alternative to `Maybe` is to use `Par` restricted to empty and singleton collections. With this encoding, the constructors for `Maybe` become `Nothing = empty`, `Just a = liftv a` (from Section 9.3), and the destructor, `maybe :: b -> (a -> b) -> Maybe a -> b` in Haskell, is defined in Encore as `def maybe(c: b, f: a->b, x: Maybe a) = otherwise(pmap(f, x), \() -> c)`.

A key omission from this list is any operation that actually treats a parallel collection in a sequential fashion. For instance, getting the first (leftmost) element is not possible. This limitation is in place to discourage sequential programming with parallel types.

9.3 From sequential to parallel types

A number of functions lift sequential types to parallel types to initiate parallel computation or dataflow.

- `liftv :: t -> Par t` converts a value to a singleton parallel collection.
- `liftf :: Fut t -> Par t` converts a future to a singleton parallel collection (following the `Par t` ≈ `[Fut t]` analogy above).
- `each :: [t] -> Par t` converts an array to a parallel collection.

One planned extension to provide a better integration between active objects and parallel collections is to allow fields to directly store collections but not as a data type but, effectively, as a parallel type. Then applying parallel operations would be more immediate.

9.4 ... and back again

A number of functions are available for getting values out of parallel types. Here is a sample:

- `select :: Par t -> Maybe t`, described in Section 9.2, provides a way of getting a single element from a collection (if present).
- `sum :: Par Int -> Int` and other fold/reduce-like functions provide operations such as summing the collection of integers.
- `sync :: Par t -> [t]` synchronises the parallel computation and produces a sequential array of the results.
- `wsync :: Par t -> Fut [t]` same as `sync`, but instead creates a computation to do the synchronisation and returns a future to that computation.

9.5 Example

The following code illustrates parallel types and combinators. It computes the total sum of all bank accounts in a bank that contain more than 10,000 euros.

The program starts by converting the sequential array of customers into a parallel collection. From this point on it applies parallel combinators to get the accounts, then the balances for these accounts, and to filter the corresponding values. The program finishes by computing a sum of the balances, thereby moving from the parallel setting back to the sequential one.

```
import party

class Main
  bank : Bank
  def main(): void {
    let
      customers = each(bank.get_customers()) -- get customers objects
      balances =
        filter(\(x: int) -> { x > 10000 }, -- filter accounts
           pmap(\(x: Account) -> x.get_balance()), -- get all balances
              pbind(customers,
                    \(x : Customer) -> x. get_accounts())) -- get all accounts
    in
      print("Total: {}\n", sum(balances))
  }
```

9.6 Implementation

At present, parallel types and combinators are implemented in Encore as a library and the implementation does not deliver the desired performance. In the future, `Par t` will be implemented as an abstract type to give the compiler room to optimise how programs using parallel combinators are translated into C. Beyond getting the implementation efficient, a key research challenge that remains to be addressed is achieving safe interaction between the parallel combinators and the existing active object model using capabilities.

10 Capabilities

The single thread of control abstraction given by active objects enables sequential reasoning inside active objects. This simplifies programming as there is no

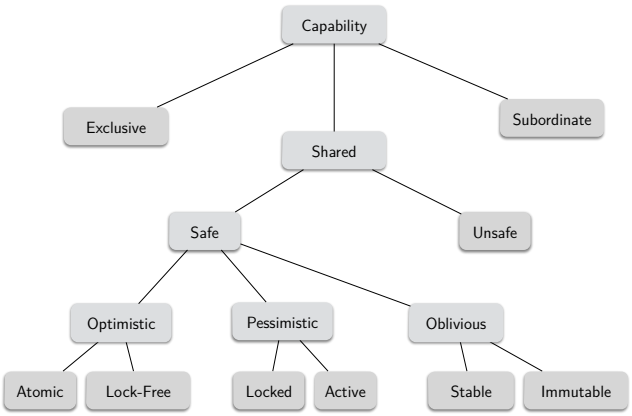


Figure 2. The hierarchy of capabilities. Leaf nodes denote concrete capabilities, non-leaves categorise.

interleaving of operations during critical sections of a program. However, unless proper encapsulation of passive objects is in place, mutable objects might be shared across threads, effectively destroying the single thread of control.

A simple solution to this problem is to enforce deep copying of objects when passing them between active objects, but this can increase the cost of message sending. (This is the solution adopted in the formal semantics of ENCORE presented in Section 12.) Copying is, however, not ideal as it eliminates cases of benign sharing of data between active objects, such as when the shared data is immutable. Furthermore, with the creation of parallel computation inside active objects using the parallel combinators of Section 9, more fine-grained ways of orchestrating access to data local to an active object is required to avoid race conditions.

These are the problems addressed by the *capability type system* in ENCORE.[8]

10.1 Capabilities for Controlling of Sharing

A *capability* is a token that governs access to a certain resource [28]. In an attempt to re-think the access control mechanisms of object-oriented programming systems, ENCORE uses capabilities *in place of* references and the resources they govern access to are objects, and often entire aggregates. In contrast to how references normally behave in object-oriented programming languages, capabilities impose principles on how and when several capabilities may govern access to a common resource. As a consequence, different capabilities impose different means of alias control, which is statically enforce at compile-time.

In ENCORE capabilities are constructed from *traits*, the units of reuse from which classes can be built (*cf.* Section 4.5). Together with a *kind*, each trait forms a capability, from which composite capabilities can be constructed. A capability provides an interface, essentially the methods of the corresponding trait. The capability's kind controls how this interface can be accessed with respect to avoiding data-races. Capabilities are combined to form classes, just as traits do, which means that the range of capabilities are specified at class definition, rather than at object creation time.

From an object's type, it is immediately visible whether it is *exclusive* to a single logical thread of control (which trivially implies that accesses to the object are not subject to data races), *shared* between multiple logical threads (in which case freedom from data races must be guaranteed by some concurrency control mechanism), or *subordinate* to some other object which protects it from data races—this is the default capability of a passive class in ENCORE. The *active* capability is the capability kind of active classes in ENCORE. Figure 2 shows the different capabilities considered, which will be discussed below.[9]

[8] Note that at the time of writing, the capability system has not been fully implemented.

[9] ENCORE may eventually *not* include all kinds of capabilities presented here, this is a matter under consideration.

10.2 Exclusive Capabilities

Exclusive capabilities are exclusive to a single thread of control. Exclusive capabilities implement a form of *external uniqueness* [12] where a single pointer is guaranteed to be the only *external* pointer to an entire aggregate. The uniqueness of the external variable is preserved by destructive reads, that is, the variable must be nullified when read unless it can be guaranteed that the two aliases are not visible to any executing threads at the same time.

Exclusive capabilities greatly simplify ownership transfer—passing an exclusive capability as an argument to a method on another active object requires the nullification of the source variable, which means that all aliases at the source to the entire aggregate are dead and that receiver has sole access to the transferred object.

10.3 Shared Capabilities

A shared capability expresses that the object is shared and, in contrast to exclusive capabilities, some dynamic means is required to guarantee data-race freedom.

The semantics of concurrent accesses via a shared capability is governed by the sharing kind. The list below overviews the semantics of concurrent accesses of safe (first six) and unsafe capabilities.

Active Active capabilities denote ENCORE's active classes. They guarantee race-freedom through dynamic, pessimistic concurrency control by serialising all its inside computation.

Atomic Atomic capabilities denote object references whose access is managed by a transaction. Concurrent operations either commit or rollback in a standard fashion. From a race-freedom perspective, atomic capabilities can be freely shared across ENCORE active objects.

An interesting question arises when considering the interaction between transactions and asynchronous message passing: can asynchronous messages "escape" a transaction? Since asynchronous messages are processed by other logical threads of control, they may be considered a side-effect that is impossible to roll-back. Some ways of resolving this conundrum are:

1. Forbidding asynchronous message sends inside transactions. Problem: *Restricts expressivity.*
2. Delaying the delivery of asynchronous message sends to commit-time of a transaction. Problem: *Reduces throughput/increases latency.*
3. Accepting this problem and leaving it up to programmer's to ensure the correctness of their programs when transactions are mixed with asynchronous message passing. Problem: *Not safe.*

Immutable Immutable capabilities describe data that is always safely accessible without concurrency control. Immutability is "deep", meaning that state can be observably modified through an immutable reference, though a method in an immutable object can mutate state created within the method or of its arguments. Immutable capabilities can be freely shared across ENCORE active objects without any need for copying.

Locked Each operation via a locked capability requires prior acquisition of a lock specific for the resource. The lock can be reentrant (analogous to a synchronised method in Java), a readers-writer lock, etc. depending on desired semantics.

LockFree The implementations of behaviours for this capability must follow a certain protocol for coordinating updates in a lock-free manner. Lock-free programming is famously subtle, because invariants must be maintained at all times, not just at select commit-points. As part of the work on ENCORE, we are implementing a type system that enforces such protocol usage on lock-free capabilities [8].

Stable Stable capabilities present an immutable view of otherwise mutable state. There are several different possible semantics for stable capabilities: *read-only references*—capability cannot be used to modify state, but it may witness changes occurring elsewhere; *fractional permissions*—if a stable capability is available, no mutable alias to an overlapping state will be possible, thereby avoiding read-write races; or *readers-writer locks*—a static guarantee that a readers-writer lock is in place and used correctly.

Unsafe As the name suggest, unsafe capabilities come with no guarantees with respect to data races. Allowing unsafe capabilities is optional, but they may be useful to give a type to embedded C code.

10.4 Subordinate Capabilities

A subordinate capability is a capability that is dominated by an exclusive or shared capability, which means that the dominating capability controls access to the subordinate. In ENCORE, passive objects are subordinate capabilities by default, meaning they are encapsulated by their enclosing active object. This corresponds to the fact that there can be no "free-standing" passive objects in ENCORE, they all live on the local heap of some active object.

Encapsulation of subordinate objects is achieved by disallowing them to be passed to non-subordinate objects. A subordinate capability is in this respect similar to the `owner` annotation from ownership types [10, 39].

Some notion of borrowing can be used to safely pass subordinate objects around under some conditions [13].

10.5 Polymorphic Concurrency Control

Capabilities allow for polymorphic concurrency control through the abstract capabilities *shared*, *safe*, *optimistic*, *pessimistic* and *oblivious*. This allows a library writer to request that a value is protected from data races by some means, but not specify those means explicitly. For example:

```
1 def transfer(from:safe Account, to:safe Account, amount:int) : void
2   to.deposit(from.withdraw(amount))
```

This expresses that the calls on `from` and `to` are safe from a concurrency standpoint. However, whether this arises from the accounts using locks, transactions or immutability is not relevant here.

Accesses through safe capabilities are interesting because the semantics of different forms of concurrency control requires a small modicum of extra work at run-time. For example, if `from` is active, then `from.withdraw()` should (implicitly) be turned into **`get`** `from.withdraw()`, or if we are *inside* an atomic capability and `to` is a locked capability, then the transfer transaction should be extended to also include `to.deposit()`, and committing the transaction involves being able to grab the lock on `to` and release it once the transaction's log has been synchronised with the object.

The exact semantics of the combinations are currently being worked out.

10.6 Composing Capabilities

A single capability is a trait plus a mode annotation. Mode annotations are the labels in Figure 2. Leaves denote concrete modes, i.e., modes that can be used in the definition of a capability or class. Remaining annotations such as safe, pessimistic etc. are valid only in types to abstract over concrete annotations, or combinations of concrete annotations.

Capabilities can be composed in three different ways: *conjunction* $C_1 \otimes C_2$, *disjunction* $C_1 \oplus C_2$, and *co-encapsulation* $C_1\langle C_2\rangle$.

A conjunction or a disjunction of two capabilities C_1 and C_2 creates a composite capability with the union of the methods of C_1 and C_2. In the case of a disjunction, C_1 and C_2 may share state without concurrency control. As a result, the same guard (whether it is linearity, thread-locality, a lock, etc.) will preserve exclusivity of the entire composite. In the case of a conjunction, C_1 and C_2 must not share state, except for state that is under concurrency control. For example, they may share a common field holding a shared capability, as long as neither capability can write the field. The conjunction of C_1 and C_2, $C_1 \otimes C_2$, can be unpacked into its two sub-capabilities C_1 and C_2, creating two aliases to the same object that can be used without regard for the other.

In contrast to conjunction and disjunction, co-encapsulation denotes a nested composition, where one capability is buried inside the other, denoted $C_1\langle C_2\rangle$. The methods of the composite $C_1\langle C_2\rangle$ are precisely those of C_1, but by exposing the nested type C_2 in the interface of the composite capability, additional operations on the type-level become available. Co-encapsulation is useful to preserve linearity of nested capabilities. For example, unless C_3 is exclusive, the capability

$C_3\langle C_1 \otimes C_2\rangle$ can be turned into $C_3\langle C_1\rangle \otimes C_3\langle C_2\rangle$ which introduces aliases to C_3 but in a way that only disjoint parts of the nested capability can be reached.

Capabilities of different kinds may be used in disjunctions and conjunctions. A capability with at least one exclusive component must be treated linearly to guarantee race-freedom of its data. A capability with at least one subordinate component will be contained inside its enclosing class. There are vast possibilities to create compositions of capabilities, and we are currently investigating their possible uses and interpretations. For example, combinations of active and exclusive capabilities allow operating on an active object as it if was a passive object until the exclusive capability is lost, after which the active object can be freely shared. This gives powerful control over the initialisation phase of an object. As another example, conjunctions of active capabilities could be used to express active objects which are able to process messages in parallel.

10.7 Implementing A Parallel Operation on Disjoint Parts of Shared State

Figures 3, 4, 5, and 6 show how the capabilities can be used to construct a simple linked list data structure of exclusive pairs, which is subsequently "unpacked" into two (logical) immutable lists of disjoint cells which are passed to different objects, and later re-packed into a single mutable list of pairs again. Support for unstructured packing and unpacking is important in ENCORE as communication across active objects has a more flexible control flow than calls on a single stack, or fork-join style parallelism.

The **`trait`** keyword introduces a new trait which requires the presence of zero or more fields in any class that includes it. Figure 3 illustrates a trait `Cell` that requires a mutable field `value` in any including class.

The compositions of cells into `WeakPair` and `StrongPair` have different reuse stories for the `Cell` trait. The cells of a `WeakPair` may be independently updated by different threads whereas the cells of a `StrongPair` always belong to the same thread and are accessed together.

For simplicity, we employ a prime notation renaming scheme for traits to avoid name clashes when a single trait is included more than once.

Figure 4 shows how three capabilities construct a singly linked list. The links in the list are subordinate objects, and the elements in the list are typed by some exclusive parameter `P`. The capabilities of the `List` class are `Add`, `Del` and `Get`. The first two are exclusive and the last is stable.

The `Add` and `Del` capabilities can add and remove exclusive `P` objects from the list. (Since these objects are exclusive, looking them up, removes them from the list to maintain linear access.) Since `Add` and `Del` share the same field `first` with the same type, they are not safe to use separately in parallel, so their combination must be a disjunction. If they had been, for example, locked capabilities instead, they would have protected their internals dynamically, so in this case, a conjunction would be allowed.

Linearity of exclusive capabilities is maintained by an explicit destructive read, the keyword `consumes`. The expression **`consume`** `x` returns the value of `x`, and updates `x` with **`null`**, logically in one atomic step.

```
1  -- declares a capability through a trait
2  trait Cell {
3    -- requirement of including class
4    require var value : int
5
6    def get() : int
7      value
8
9    def set(v : int) : void
10     value = v
11 }
12
13 // constructs class from two exclusive capabilities
14 passive class WeakPair = exclusive Cell ⊗ exclusive Cell' {
15   var value : int -- mutable field
16   var value' : int
17 }
18
19 passive class StrongPair = exclusive Cell ⊕ exclusive Cell' {
20   var value : int
21   var value' : int
22 }
```

Figure 3. Two different implementations of pairs from the same building blocks. `WeakPair` uses conjunction (⊗) and `StrongPair` disjunction (⊕).

The `Get` capability overlaps with the others, but the requirement on the field `first` is different: it considers the field immutable and its type stable through the `Iterator` capability (cf. Figure 5). As the `Get` capability's state overlaps with the other capabilities, their composition must be in the form of a disjunction.

Inside the `Get` trait, the list will not change—the `first` field cannot be reassigned and the `Iterator` type does not allow changes to the chain of links. The `Get` trait however is able to perform *reverse borrowing*, which means it is allowed to read exclusive capabilities on the heap *non-destructively* and return them, as long as they remain stack bound. The stack-bound reference is marked by a type wrapper, such as **`S`**`(P)`.

The link capabilities used to construct the list are shown in Figure 5; they are analogous to the capabilities in `List`, and are included for completeness.

Finally, Figure 6 shows the code for unpacking a list of `WeakPairs` into two logical, stable lists of cells that can be operated on in parallel. The stable capability allows multiple (in this case two) active objects to share part of the list structure with a promise that the list will not change while they are looking.

On a call to `start()` on a `Worker`, the list of `WeakPairs` is split into two in two steps. Step one (line 10–11) turns the `List` disjunction into a conjunction by *jailing* the `Add` and `Del` components which prevents their use until the list is reassembled again on line 35. Step two (line 12) turns the iterator into two by unpacking the pair into two cells.

The jail construct is used to temporarily render part of a disjunction inaccessible. In the example, `Add<WeakPair>` ⊕ `Del<WeakPair>` ⊕ `Get<Cell>` is turned

```
passive class List<exclusive P> =
  exclusive Add<P> ⊕ exclusive Del<P> ⊕ stable Get<P> {
  var first : Link<P>; // subordinate, so strongly encapsulated
}

trait Add<exclusive P> {
  require var first : Link<P>;

  def append(v : P) : void {
    var tmp : Link = new Link<P>();
    tmp.setNext(first);
    tmp.setValue(consume v); // destructive read
  }
}

trait Del<exclusive P> {
  require var first : Link<P>;

  def remove(i : int) : P {
    var tmp : Link<P> = first;
    while (i > 0) {
      i = i-1;
      tmp = tmp.getNext();
    };
    var v : P = tmp.getValue();
    ... // code for removing link omitted for brevity
    return consume v;
  }
}

trait Get<exclusive P> {
  // non-assignable ‘‘final’’ field
  require val first : Iterator<P>;

  def lookup(i : int) : S(P) { // stack-bound return
    var tmp : S(Iterator<P>) = first;
    while (i > 0) {
      i = i-1;
      tmp = tmp.next();
    };
    return tmp.value();
  }
}
```

Figure 4. A list class. `P` above is a type parameter which allows deep unpacking of the object.

```
1  passive class Link<exclusive P> = Node<P> ⊕ Iterator<P> {
2    var elmt : P;
3    var next : Link<P>;
4  }
5
6  trait Node<exclusive P> {
7    require var elmt : P;
8    require var next : Link<P>;
9
10   def setNext(n : Link<P>) : void { next = n; }
11
12   def setValue(e : P) : void { elmt = consume e; }
13
14   def getNext() : Link<P> { return next; }
15
16   def getValue() : P { return consume elmt; }
17 }
18
19 trait Iterator<exclusive P> {
20   require val elmt : P;
21   require val next : Iterator<P>;
22
23   def next() : S(Iterator<P>) { return next; }
24
25   def value() : S(P) { return elmt; }
26 }
```

Figure 5. Definition of the link class with a stable iterator capability to support non-destructive parallel iteration over elements in the list.

into `J(Add<WeakPair> ⊕ Del<WeakPair>) ⊗ Get<Cell>`. The latter type allows unpacking the exclusive reference into two, but since one cannot be used while it is jailed, the exclusivity of the referenced object is preserved.

The lists of cells are passed to two workers (itself and `other`) that perform some work, before passing the data back for reassembly (line 32).

11 Examples

A good way to get a grip on a new programming language is to study how it is applied in larger programs. To this end, three Encore programs, implementing a thread ring (Section 11.1), a parallel prime sieve (Section 11.2), and a graph generator following the preferential attachment algorithm (Section 11.3), are presented.

11.1 Example: Thread Ring

Thread Ring is a benchmark for exercising the message passing and scheduling logic of parallel/concurrent programming languages. The program is completely

```
class Worker
  var data : List<WeakPair>
  var one : J(Add<WeakPair> ⊕ Del<WeakPair>)
  var two : J(Add<WeakPair> ⊕ Del<WeakPair>) ⊗ Get<Cell>
  val other : Worker
  var sum : int

  def start() : void
    let
      var stash : J(Add<WeakPair> ⊕ Del<WeakPair>),
          iter : Get<WeakPair> = consume this.data
      var fst_i : Get<Cell>, snd_i : Get<Cell> = consume iter
    in {
      this.stash = stash; -- keep this part for later
      this ! do_work(consume fst_i, this);
      other ! do_work(consume snd_i, this);
    }

  def do_work(w : Get<Cell>, s : Worker) : void
    let
      sum = 0
    in {
      repeat i <- w.length()
        sum = sum + w.get(i).get();
      s ! done(sum);
      s ! reassemble(w);
    }

  def done(sum:int) : void
    this.sum = this.sum + sum

  def reassemble(part : Get<Cell>) : void
    if this.two == null
    then this.two = this.one + part
    else this.data = this.two + part
```

Figure 6. Parallel operations on a single list of pairs using unpack and re-packing. Note that this code would not type check for **var** a: List<StrongPair> as StrongPair is built from a disjunction that does not allow unpacking.

sequential, but deceptively parallel. The corresponding Encore program (Figure 7) creates 503 active objects, links them forming a ring and passes a message containing the remaining number of hops to be performed from one active object to the next. When an active object receives a message containing 0, it prints its own id and the program finishes.

In this example, the active objects forming the ring are represented by the class `Worker`, which has field `id` for worker's id and `next` for the next active object in the ring. Method `init` is the constructor and the ring is set up using method `setNext`. The method `run` receives the number of remaining hops, checks whether this is larger than 0. If it is, it sends an asynchronous message to the

```
class Main
  def main(): void {
    let
      index = 1
      first = new Worker(index)
      next = null : Worker
      nhops = 50*1000*1000
      ring_size = 503
      current = first
    in {
      while (index < ring_size) {
        index = index + 1;
        next = new Worker(index);
        current!setNext(next);
        current = next;
      };
      current!setNext(first);
      first!run(nhops);
    }
  }

class Worker
  id : int
  next : Worker

  def init(id : int): void
    this.id = id

  def setNext(next: Worker): void
    this.next = next

  def run(hops : int): void
    if (hops > 0) then
      this.next!run(hops-1)
    else
      print(this.id)
```

Figure 7. Thread Ring Example

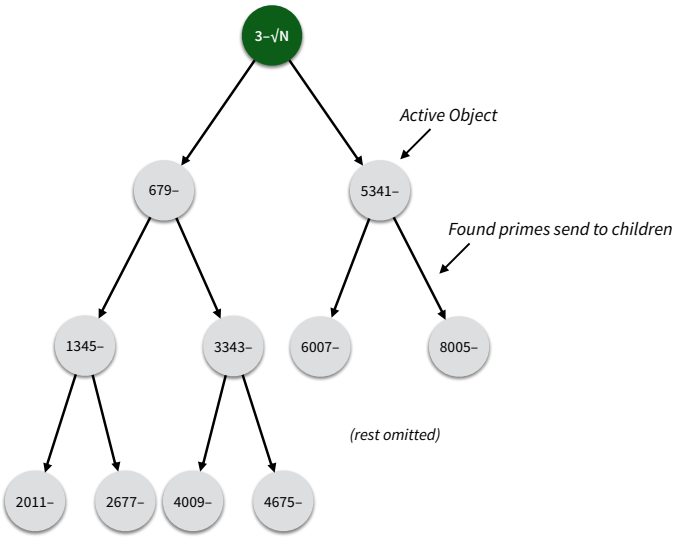


Figure 8. Overview of the parallel prime sieve. The root object finds all primes in $[2, \sqrt{N}]$ and broadcasts these to filter objects that cancel all multiples of these in some ranges. When a filter receives the final "done" message, it will scan its range for remaining (prime) numbers and report these to an object that keeps a tally.

next active object with the number of remaining hops decrement by 1. Otherwise, the active object has finished and prints its id.

11.2 Example: Prime Sieve

This example considers an implementation of the parallel Sieve of Eratosthenes in Encore. Recall that the Sieve works by filtering out all non-trivial multiples of 2, 3, 5, etc., thereby revealing the next prime, which is then used for further filtering. The parallelisation is straightforward: one active object finds all primes in $\sqrt{N}$ and uses M filter objects to cancel out all non-primes in (chunks of) the interval $[\sqrt{N}, N]$. An overview of the program is found in Figure 8 and the code is spread over Figures 9, 10 11 and 12. Which each filter object finally receives a "done" message, they scan their ranges for remaining (prime) numbers and report these to a special reporter object that keeps a tally of the total number of primes found.

The listing of the prime sieve program starts by importing libraries. The most important library component is a bit vector data type, implemented as a thin Encore wrapper around a few lines of C to flip individual bits in a vector. Figure 9 shows the class of the reporter object that collects and aggregates the results of all filter active objects. When it is created, is it told how many candidates are considered (e.g., all the primes in the first 1 billion natural numbers), and as

```
1  import lib
2
3  class Reporter
4    primes:int
5    candidates:int
6
7    def init(c:int) : void
8      this.candidates = c
9
10   def report(p:int, c:int) : void {
11
12     this.candidates = this.candidates - c;
13     this.primes = this.primes + p;
14
15     if this.candidates == 0 then print this.primes;
16   }
```

Figure 9. Prime Sieve (a). The `Reporter` class collects the reports from all filter active objects and summarises the (number of) primes found.

every filter reports in, it reports the number of primes found in the number of candidates considered.

The main logic of the program happens in the `Filter` class. The filter objects form a binary tree, each covering a certain range of the candidate numbers considered. The lack of a math library requires a home-rolled power function (`pow()` below) and an embedded C-level `sqrt()` function (Lines 105–106).

The main filter calls the `found_prime()` function with a prime number. This causes the program to propagate the number found to its children (line 77). This allows them to process the number in parallel with the active object doing a more lengthy operation in `cancel_one()`, namely iterating over its bit vector and cancelling out all multiples of the found prime.

Once the main active object has found all the primes in $\sqrt{N}$, it calls `root_done()` which is propagated in a similar fashion as `found_prime()`. Finally, the `done()` method is called on each filter active object, which scans the bit vector for any remaining numbers that have not been cancelled out. Those are the prime numbers which are sent to the reporter.

11.3 Example: Preferential Attachment

Preferential attachment is a model of graph construction that produces graphs whose node degrees have a power law distribution. Such graphs model a number of interesting phenomena such as the growth of the World Wide Web or social networks [3, 6].

The underlying idea of preferential attachment is that nodes are added incrementally to a graph by establishing connections with existing nodes, such that edges are added from each new node to a random existing node with probability proportional to the degree distribution of the existing nodes. Consequently, the

```
class Filter
  bitset:Bitset
  start:int
  stop:int
  first:int
  report:Reporter
  adjust:int
  left:Filter
  right:Filter

  def pow(n:int, to:int) : int
    if to == 0 then 1 else n * this.pow(n, to - 1)

  def init(start:int, stop:int, report:Reporter, depth:int) : void {
    let
      size = (stop - start) / (this.pow(2, depth) - 1)
      adjusted_size = if depth > 1 then ((size + 99) / 100 ) * 100 else stop - start
    in
      this.bitset = new Bitset(adjusted_size);
    this.start = start;
    this.stop = start + this.bitset.size;
    this.report = report;
    this.adjust = if start % 2 == 0 then 1 else 0;

    -- print("new Filter({}, {})\n", this.start, this.stop);

    if depth > 1 then
      let
        half = ((((stop - this.stop) / 2) + 99) / 100 ) * 100
      in
        {
          this.left = new Filter(this.stop, stop - half, report, depth - 1);
          this.right = new Filter(stop - half, stop, report, depth - 1);
        }
  }

  def calculate_offset(start:int, prime:int) : int
    let
      m = prime * prime - start
      p = if start % prime == 0 then 0 else ((start / prime) * prime + prime) - start
    in
      if p < m then m else p
```

Figure 10. Prime Sieve (b). The `Filter` class (continued in next figure) is the main work horse of this program.

```
61  def cancel_one(prime:int) : void
62    let
63      stop = this.bitset.size
64      i = this.calculate_offset(this.start, prime)
65    in {
66      while i < stop
67        {
68          this.bitset.unset(i);
69          i = i + prime;
70        };
71    }
72
73  def has_children() : bool
74    this.left == null
75
76  def found_prime(p:int) : void {
77    if this.has_children() then { this.left ! found_prime(p); this.right ! found_prime(p); };
78    this.cancel_one(p);
79  }
80
81  def root_done() : void {
82    if this.has_children() then { this.left ! root_done(); this.right ! root_done(); };
83    this.done();
84  }
85
86  def done() : void
87    let
88      i = this.adjust
89      j = this.bitset.size
90      primes = 0
91    in
92      {
93        while i < j
94          {
95            if this.bitset.isset(i) then primes = primes + 1;
96            i = i + 2;
97          };
98        this.report ! report(primes, this.bitset.size);
99      }
```

Figure 11. Prime Sieve (c). The `Filter` class (continued from previous figure) is the main work horse of this program.

```
101 class Main
102   def round_up(i:int) : int
103     ((i + 99) / 100 ) * 100
104
105   def sqrt(i:int) : int
106     embed int sqrt(#{i}); end
107
108   def main() : void
109     let
110       candidates = 1000000000
111       depth = 7
112       stop = this.round_up(this.sqrt(candidates))
113       array = new Bitset(stop)
114       guard = new Reporter(candidates)
115       num = 3
116       idx = 2
117       primes = 1
118       root = new Filter(stop, candidates, guard, depth)
119     in {
120       while num < stop
121         {
122           if array.isset(idx) then {
123             primes = primes + 1;
124             root ! found_prime(num);
125             let
126               j = (num * num) - 1
127             in
128               while j < stop
129                 {
130                   array.unset(j);
131                   j = j + num;
132                 };
133           };
134           num = num + 2;
135           idx = idx + 2;
136         };
137       guard ! report(primes, stop);
138       root ! root_done();
139     }
```

Figure 12. Prime Sieve (d). The `Main` class sets up the program and finds all the primes in the first $\sqrt{N}$ (here hard coded to 1 billion) candidates.

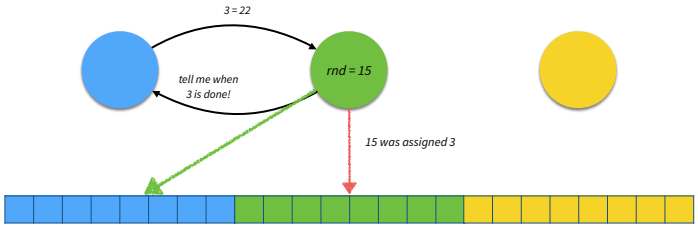


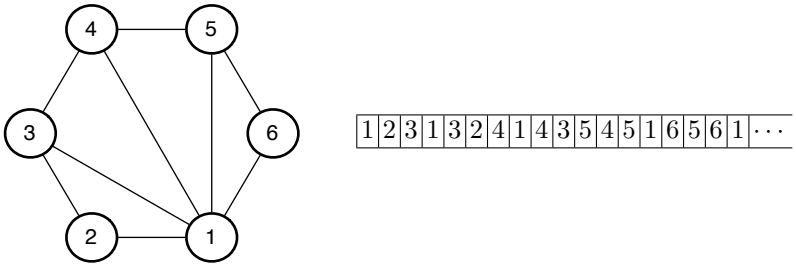
Figure 13. High-level design of the parallel preferential attachment.

better connected a node is, the higher the chance new nodes will be connected to it (thereby increasing the chance in the future that more new nodes will be connected to it).

The preferential algorithm is based on two parameters: n the total number of nodes in the final graph and k the number of unique edges connecting each newly added node to the existing graph. A sequential algorithm for preferential attachment is:

1. Create a fully connected graph of size k (the clique). This is a completely deterministic and all the nodes in the initial graph will be equally likely to be connected to by new nodes.
2. For $i = k+1, \ldots, n$, add a new node n_i, and randomly select k *distinct* nodes from $n_1, \ldots, n_{i-1}$ with probability proportional to the degree of the selected node and add the edges from n_i to the selected nodes to the graph.

One challenge is handling the probabilities correctly. This can be done by storing the edges in an array of size $\approx n \times k$, where every pair of adjacent elements in the array represents an edge. As an example, consider the following graph and its encoding.



The number of times a node appears in this array divided by the size of the array is precisely the required probability of selecting the node. Thus, when adding new edges, performing a uniform random selection from this array is sufficient to select target nodes.

In the implementation a simple optimisation is made. Half of the edge information is statically known—that is (ignoring the clique), for each $n > k$, the array will look like the following:

$\cdots$	n	$m_{n,1}$	n	$m_{n,k}$	$\cdots$	n	$m_{n,k}$	$\cdots$

The indices where the edges for node n will be stored can be calculated in advance, and thus these occurrences of n need not be stored, and the amount of space required can be halved.

Parallelising preferential attachment is non-trivial due to the inherent temporality in the problem: the correctness (with respect to distinctness) of all random choices for an addition depends on the values selected for earlier nodes. However, even though a node may not yet appear in some position in the array, it is possible to compute in parallel a candidate for the desired for *all* future nodes. Then these can gradually be fulfilled (out of order) and the main challenge is ensuring that distinctness of edges is preserved. This is done by checking whenever new edges are added and randomly selecting again when a duplicate edge is added.

The naive implementation shown here attempts to parallelise the algorithm by creating A active objects each responsible for the edges of some nodes in the graph. Every active object proceeds according to the algorithm above, but with non-overlapping start and stop indexes. If a random choice picks an index that is not yet filled in, a message is sent to the active object that owns that part of the array with a request to be notified when that information becomes available.

The requirement that edges are *distinct* needs to be checked whenever a new edge is added. If a duplicate is found, the algorithm just picks another random index. With reasonably large graphs (say 1 million nodes with 20 edges each), duplicate edges is rare, but the scanning is still necessary, and this is more costly in the parallel implementation compared to the sequential one, because in the sequential algorithm all edges are available at the time the test for duplicates is made, but this is not the case in the parallel algorithm.

Figure 13 shows a high-level overview of the implementation. The colour-coded workers own the write rights to the equi-coloured part of the array. A green arrow denotes read rights, a red array denotes write rights. The middle worker attempts to the edge at index 3 in the array, which is not yet determined. This prompts a request to the (left) worker that owns the array chunk to give this information once the requested value becomes available. Once this answer is provided, the middle active object writes this edge into the correct place, provided it is not a duplicate, and forwards the results to any active object that has requested it before this point.

The Encore implementation of preferential attachment is fairly long and can be found in Appendix A.

12 Formal Semantics

This section presents the semantics of a fragment of Encore via a calculus called μEncore. The main aim of μEncore is to formalise the proposed concurrency model of Encore, and thereby establish a formal basis for the development of Encore and for research on type-based optimisations in the UpScale project. μEncore maintains a strong notion of locality by ensuring that there is no shared data between different active objects in the system.

Syntactic categories.
x in Variable
e in Expression
v in Value

Definitions.

$$
\begin{aligned}
P &::= \overline{IF}\ \overline{CL}\ e \\
T &::= \mathsf{bool} \mid I \mid C \mid \mathsf{void} \mid \mathsf{Fut}\ T \mid \overline{T} \to T \\
IF &::= [\texttt{passive}]\ \texttt{interface}\ I\,\{\,[\overline{Sg}]\,\} \\
CL &::= [\texttt{passive}]\ \texttt{class}\ C\,\{\,[\overline{x} : \overline{T};]\,\overline{M}\} \\
Sg &::= \texttt{def}\ m\,([\overline{x} : \overline{T}]) : T \\
M &::= Sg\ \{\ e\ \} \\
e &::= e; e \mid x \mid v \mid x.x \mid \texttt{skip} \mid x = e \mid \texttt{while}\ e\,\{\,e\,\} \mid \texttt{await}\ e \\
&\quad \mid \texttt{if}\ e\ \{\,e\,\}\ \texttt{else}\,\{\,e\,\} \mid \texttt{let}\ \overline{x} = \overline{e}\ \texttt{in}\ \{\,e\,\} \mid \texttt{suspend} \\
&\quad \mid e \diamond m(\overline{e}) \mid e \lightning m(\overline{e}) \mid \texttt{new}\ C\,(\overline{e}) \mid \texttt{get}\ e \mid e \rightsquigarrow e \\
&\quad \mid \texttt{spore}\ \overline{x} = \overline{e}\ \texttt{in}\ \lambda(\overline{x} : \overline{T}) \to e : T \mid e(\overline{e}) \\
v &::= \texttt{true} \mid \texttt{false} \mid \texttt{()}
\end{aligned}
$$

Figure 14. Syntax of μENCORE. Terms like $\overline{e}$ and $\overline{x}$ denote (possibly empty) lists over the corresponding syntactic categories, square brackets `[]` denote optional elements.

12.1 The Syntax of μEncore

The formal syntax of μENCORE is given in Figure 14. A program P consists of interface and class declarations followed by an expression e which acts as the main block.[10] The types T includes Booleans bool (ignoring other primitive types), a type void (for the `()` value), type Fut T for futures, interfaces I, passive classes C, and function types $\overline{T} \to T$. In μENCORE, objects can be active or passive. *Active* objects have an independent thread of execution. To store or transfer local data, an active object uses *passive* objects. For this reason, interfaces *IF* and classes *CL* can be declared as **`passive`**. In addition, an interface has a name I and method signatures *Sg* and class has a name C, fields $\overline{x}$ of type $\overline{T}$, and methods $\overline{M}$. A method signature *Sg* declares the return type T of a method with name m and formal parameters $\overline{x}$ of types $\overline{T}$. M defines a method with signature *Sg*, and expressions e. When constructing a new object of a class C by a statement **`new`** $C(\overline{e})$, the new object may be either active or passive (depending on the class).

Expressions include, variables (local variables and fields of objects), values, sequential composition $e_1; e_2$, assignment, **`skip`** (to make semantics easier to write), **`if`**, **`let`**, and **`while`** constructs. Expressions may access the fields of an object, the method's formal parameters and the fields of other objects. *Values* v are expressions on a normal form, *let*-expressions introduce local variables. Cooperative scheduling in μENCORE is achieved by explicitly suspending the execution of the active stack. The statement **`suspend`** unconditionally suspends the execution of the active method invocation and moves this to the queue. The statement **`await`** e conditionally suspends the execution; the expression e evaluates to a future f, and execution is suspended only if f has not been fulfilled.

Communication and *synchronisation* are decoupled in μENCORE. In contrast to ENCORE, μENCORE makes explicit in the syntax the two kinds of method call: $e \diamond m(\overline{e})$ corresponds to a synchronous call and $e \lightning m(\overline{e})$ corresponds to an

[10] μENCORE supports interfaces, though ENCORE does not yet. ENCORE with combine interfaces with traits.

asynchronous call. Communication between active objects is based on asynchronous method calls $o!m(\overline{e})$ whose return type is $\mathsf{Fut}\ T$, where T corresponds to the return type of the called method m. Here, o is an object expression, m a method name, and $\overline{e}$ are expressions providing actual parameter values for the method invocation. The result of such a call is a future that will hold the eventual result of the method call. The caller may proceed with its execution *without blocking*. Two operations on futures control synchronisation in μENCORE. The first is `await` f, which was described above. The second is `get` f which retrieves the value stored in the future when it is available, blocking the active object until it is. Futures are first-class citizens of μENCORE. Method calls on passive objects may be synchronous and asynchronous. Synchronous method calls $o \diamond m(\overline{e})$ have a Java-like reentrant semantics. Self calls are written `this`$!m(\overline{e})$ or `this`$\diamond m(\overline{e})$.

Anonymous functions and *future chaining* Anonymous functions are available in μENCORE in the form of spores [27]. A spore `spore` $\overline{x}' = \overline{e}'$ `in` $\lambda(\overline{x} : \overline{T}) \to e : T$ is a form of closure in which the dependencies on local state $\overline{e}'$ are made explicit; i.e., the body e of the lambda-function does not refer directly to variables outside the spore. Spores are evaluated to create closures by binding the variables $\overline{x}'$ to concrete values which are controlled by the closure. This ensures race-free execution even when the body e is not pure. The closure is evaluated by function application $e(\overline{e})$ where the arguments $\overline{e}$ are bound to the variables $\overline{x}$ of the spore, before evaluating the function body e. Closures are first class values. *Future chaining* $e_1 \rightsquigarrow e_2$ allows the execution of a closure e_2 to be spawned into a parallel task, triggered by the fulfilment of a future e_1.

12.2 Typing of μEncore

Typing judgments are on the form $\Gamma \vdash e : T$, where the typing context Γ maps variables x to their types. (For typing the constructs of the dynamic semantics, Γ will be extended with values and their types.) Write $\Gamma \vdash \overline{e} : \overline{T}$ to denote that $\Gamma \vdash e_i : T_i$ for $1 \leq i \leq |\overline{e}|$, assuming that $|\overline{e}| = |\overline{T}|$. Types are not assigned to method definitions, class definitions and the program itself; the corresponding type judgements simply express that the constructs are internally well-typed by a tag "`ok`".

Auxiliary definitions. Define function $\mathit{typeOf}(T, m)$ such that: (1) $\mathit{typeOf}(T, m) = \overline{T} \to T'$ if the method m is defined with signature $\overline{T} \to T'$ in the class or interface T; (2) $\mathit{typeOf}(T, x) = T'$ if a class T has a field x declared with type T'; and (3) $\mathit{typeOf}(C) = \overline{T} \to C$ where $\overline{T}$ are the types of the constructor arguments. Further define a predicate $\mathit{active}(T)$ to be true for all active classes and interfaces T and false for all passive classes and interfaces. By extension, let $\mathit{active}(o) = \mathit{active}(C)$ if o is an instance of C.

Subtyping. Let class names C of active classes also be types for the type analysis and let $\preceq$ be the smallest reflexive and transitive relation such that

- $T \preceq \mathsf{void}$ for all T,
- $C \preceq I \iff \forall m \in I \cdot \mathit{typeOf}(C, m) \preceq \mathit{typeOf}(I, m)$
- $\overline{T} \preceq \overline{T}' \iff n = \mathit{length}(\overline{T}) = \mathit{length}(\overline{T}')$ and $T_i \preceq T_i'$ for $1 \leq i \leq n$

$$\frac{}{\Gamma \vdash () : \mathsf{void}}\ (\text{T-Void}) \qquad \frac{}{\Gamma \vdash \texttt{true} : \mathsf{bool}}\ (\text{T-True}) \qquad \frac{}{\Gamma \vdash \texttt{false} : \mathsf{bool}}\ (\text{T-False})$$

$$\frac{\Gamma \vdash e : \mathsf{Fut}\ T}{\Gamma \vdash \texttt{await}\ e : \mathsf{void}}\ (\text{T-Await}) \qquad \frac{\Gamma \vdash e : \mathsf{Fut}\ T}{\Gamma \vdash \texttt{get}\ e : T}\ (\text{T-Get}) \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ (\text{T-Var1}) \qquad \frac{\Gamma(v) = T}{\Gamma \vdash v : T}\ (\text{T-Val}) \qquad \frac{\Gamma \vdash x : T' \quad T' \preceq T}{\Gamma \vdash x : T}\ (\text{T-Sub})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{void} \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1; e_2 : T}\ (\text{T-Seq}) \qquad \frac{\Gamma \vdash \overline{e} : \overline{T}}{\Gamma \vdash e(\overline{e}) : T}\ (\text{Application}) \qquad \frac{\Gamma \vdash e' : \mathtt{Fut}\ T' \quad \Gamma \vdash e : T' \to T}{\Gamma \vdash e' \rightsquigarrow e : \mathtt{Fut}\langle T \rangle}\ (\text{Fut-Chain}) \qquad \frac{\Gamma \vdash e : T \quad \Gamma(x) = T}{\Gamma \vdash x = e : T}\ (\text{T-Assign}) \qquad \frac{\Gamma(x_1) = C \quad \mathit{typeOf}(C, x_2) = T}{\Gamma \vdash x_1.x_2 : T}\ (\text{T-Var2})$$

$$\frac{\Gamma \vdash \overline{x} = \overline{e}_1 : \overline{T}_1 \quad \Gamma, \overline{x} \mapsto \overline{T}_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{let}\ \overline{x} = \overline{e}_1\ \texttt{in}\ \{\, e_2 \,\} : T_2}\ (\text{T-Let}) \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \mathsf{void}}{\Gamma \vdash \texttt{while}\ e_1 \{\, e_2 \,\} : \mathsf{void}}\ (\text{T-While}) \qquad \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \texttt{if}\ e\ \{\, e_1 \,\}\ \texttt{else}\ \{\, e_2 \,\} : T}\ (\text{T-Cond})$$

$$\frac{\mathit{typeOf}(T', m) = \overline{T} \to T \quad \Gamma \vdash e : T' \quad \Gamma \vdash \overline{e} : \overline{T}}{\Gamma \vdash e \sharp m(\overline{e}) : \mathsf{Fut}\ T}\ (\text{T-AsyncCall}) \qquad \frac{\mathit{typeOf}(T', m) = \overline{T} \to T \quad \Gamma \vdash e : T' \quad \Gamma \vdash \overline{e} : \overline{T} \quad \neg \mathit{active}(T')}{\Gamma \vdash e \diamond m(\overline{e}) : T}\ (\text{T-SyncCall}) \qquad \frac{\Gamma \vdash \overline{e} : \overline{T} \quad \mathit{typeOf}(C) = \overline{T} \to C}{\Gamma \vdash \texttt{new}\ C\ (\overline{e}) : C}\ (\text{T-New})$$

$$\frac{\Gamma \vdash \overline{e}' : \overline{T}' \quad \overline{x}' : \overline{T}', \overline{x} \mapsto \overline{T} \vdash e : T}{\Gamma \vdash \texttt{spore}\ \overline{x}' = \overline{e}'\ \texttt{in}\ \lambda(\overline{x} : \overline{T}) \to e : T\ \ \overline{T} \to T}\ (\text{T-Spore}) \qquad \frac{\Gamma, \overline{x} \mapsto \overline{T} \vdash e : T}{\Gamma \vdash \texttt{def}\ m\ (\overline{x} : \overline{T}) : T\ \{\ e\ \}\ \texttt{ok}}\ (\text{T-Method})$$

$$\frac{\Gamma' \vdash M\ \texttt{ok for all}\ M \in \overline{M} \quad \Gamma' = \Gamma, \overline{a}_1 \mapsto \overline{T}_1, \texttt{this} \mapsto C}{\Gamma \vdash \texttt{class}\ C\ \{\, \overline{a}_1 : \overline{T}_1;\ \overline{M} \,\}\ \texttt{ok}}\ (\text{T-Class}) \qquad \frac{\Gamma \vdash e : \mathsf{void} \quad \Gamma \vdash CL\ \text{for all}\ CL \in \overline{CL}}{\Gamma_v \vdash \overline{IF}\ \overline{CL}\ e\ \texttt{ok}}\ (\text{T-Program})$$

$$\Gamma \vdash \texttt{suspend} : \mathsf{void}\ (\text{T-Suspend}) \qquad \Gamma \vdash \texttt{skip} : \mathsf{void}\ (\text{T-Skip})$$

Figure 15. The type system for μEncore.

– $T_1 \to T_2 \preceq T_1' \to T_2' \iff T_1' \preceq T_1 \wedge T_2 \preceq T_2'$

A type T is a subtype of T' if $T \preceq T'$. The typing system of μEncore is given in Figure 15 and is mostly be standard. Note that rule T-Spore enforces that all dependencies to the local state to be explicitly declared in the spore.

12.3 Semantics of μEncore

The semantics of μEncore is presented as an operational semantics in a context-reduction style [16], using a multi-set of terms to model parallelism (from from rewriting logic [26]).

Run-time Configurations. The run-time syntax of μEncore is given in Figure 16. A *configuration cn* is a multiset of active objects (plus a local heap

of passive objects per active object), method invocation messages and futures. The associative and commutative union operator on configurations is denoted by whitespace and the empty configuration by ε.

An active object is given as a term $g(\mathit{active}, \mathit{hp}, q)$ where *active* is the method invocation that is executing or *idle* if no method invocation is active, *hp* a multiset of objects, and q is a queue of suspended (or not yet started) method invocations.

An *object obj* is a term $o(\sigma)$ where o is the object's identifier and σ is an assignment of the object's fields to values. Concatenation of such assignments is denoted $\sigma_1 \circ \sigma_2$.

In an *invocation message* $m(o, \overline{v}, \mathit{hp}, f)$, m is the method name, o the callee, $\overline{v}$ the actual parameter values, *hp* a multiset of objects (representing the data transferred with the message), and f the future that will hold the eventual result. For simplicity, the futures of the system are represented as a mapping *fut* from future identifiers f to either values v or $\bot$ for unfulfilled futures.

The queue q of an active object is a sequence of method invocations (*task*). A *task* is a term $t(\mathit{fr} \circ \mathit{sq})$ that captures the state of a method invocation as sequence of stack frames $\{\sigma|e\}$ or $\{\sigma|E\}$ (where E is an evaluation context, defined in Figure 18), each consisting of bindings for local variables plus either the expression being run for the active stack frame or a continuation (represented as evaluation context) for blocked stack frames. *eos* indicates the bottom of the stack. Local variables also include a binding for `this`, the target of the method invocation. The bottommost stack frame also includes a binding for variable *destiny* to the future to which the result of the current call will be stored.

Expressions e are extended with a polling operation $e?$ on futures that evaluates to `true` if the future has been fulfilled or `false` otherwise. Values v are extended with identifiers for the dynamically created objects and futures, and with closures. A *closure* is a dynamically created value obtained by reducing a spore-expression (after sheep cloning the local state). Further assume for simplicity that *default*(T) denotes a default value of type T; e.g., `null` for interface and class types. Also, classes are not represented explicitly in the semantics, but may be seen as static tables of field types and method definitions. Finally, the run-time type Closure marks the run-time representation of a closure. To avoid introducing too many new run-time constructs, closures are represented as active objects with an empty queue and no fields.

The *initial configuration* of a program reflects its main block. Let o be an object identifier. For a program with main block e the initial configuration consists of a single dummy active object with an empty queue and a task executing the main block itself: $g(t(\{\mathit{this} \mapsto \langle \mathsf{Closure}, o\rangle|e\} \circ \mathrm{eos}), o(\varepsilon), \emptyset)$.

A Transition System for Configurations. Transition rules transform configurations *cn* into new configurations. Let the reflexive and transitive transition relation $\rightarrow$ capture transitions between configurations. A run is a possibly terminating sequence of configurations $\mathit{cn}_0, \mathit{cn}_1, \ldots$ such that $\mathit{cn}_i \rightarrow \mathit{cn}_{i+1}$. Rules apply to subsets of configurations (the standard context rules for configurations are not listed). For simplicity we assume that configurations can be reordered to

$$
\begin{array}{rcl}
cn & ::= & \varepsilon \mid msg \mid fut \mid group \mid cn\ cn \\
group & ::= & g(active, hp, q) \\
hp & ::= & \varepsilon \mid obj \mid hp \cup hp \\
obj & ::= & o(\sigma) \\
msg & ::= & m(o, \overline{v}, hp, f) \\
fut & ::= & \varepsilon \mid fut[f \mapsto v] \mid fut[f \mapsto \bot] \\
q & ::= & \emptyset \mid task \mid q \cup q \\
task & ::= & t(fr \circ sq)
\end{array}
\qquad
\begin{array}{rcl}
\sigma & ::= & \varepsilon \mid x \mapsto \langle T, v \rangle \mid \sigma \circ \sigma \\
active & ::= & task \mid idle \\
fr & ::= & \{\sigma | e\} \\
bfr & ::= & \{\sigma | E\} \\
sq & ::= & bfr \circ sq \mid \text{eos} \\
v & ::= & \ldots \mid o \mid f \mid unit \\
 & \mid & \texttt{closure}\ \overline{x} = \overline{v}\ \texttt{in}\ \lambda(\overline{x} : \overline{T}) \to e : T \\
e & ::= & \ldots \mid e? \\
T & ::= & \ldots \mid \mathsf{Closure}
\end{array}
$$

Figure 16. Run-time syntax; here, o and f are identifiers for objects and futures, and x is the name of a variable.

match the left hand side of the rules, i.e., matching is modulo associativity and commutativity as in rewriting logic [26].

Auxiliary functions. If the class of an object o has a method m, let $\mathit{bind}(m, o, \overline{v})$ and $\mathit{abind}(m, o, \overline{v}, f)$ return a frame resulting from the activation of m on o with actual parameters $\overline{v}$. The difference between these two functions is that the *abind* introduces a local variable *destiny* bound to $\langle T, f \rangle$ where T is the return type of the frame. If the binding succeeds, the method's formal parameters are bound to $\overline{v}$. The function $\mathit{select}(q, \mathit{fut})$ schedules a task which is ready to execute from the task queue q which belongs to an active object g with $g(\mathit{idle}, hp, q)$. The function $\mathit{atts}(C, o)$ returns the initial field assignment σ of a new instance o of class C in which the fields are bound to default values. The function $\mathit{init}(C)$ returns an activation of the *init* method of C, if defined. Otherwise it returns the empty task $\{\varepsilon | ()\}$. The predicate $\mathit{fresh}(n)$ asserts that a name n is globally unique (where n may be an identifier for an object or a future). The definition of these functions is straightforward but requires that the class table is explicit in the semantics, which we have omitted for simplicity.

Sheep cloning. Given an object identifier v and a heap hp, let $\mathit{lookup}(v, hp)$ return the object corresponding to v in hp. Given a list of object identifiers $\overline{v}$ and a heap hp, let $\mathit{rename}(\overline{v}, hp, \sigma)$ return a mapping that renames all passive objects reachable from $\overline{v}$ in hp. Given a list of object identifiers $\overline{v}$ and a heap hp, $\mathit{copy}(\overline{v}, hp, \mathit{transfer})$ returns the sub-heap reachable from $\overline{v}$ in hp. The formal definitions of these functions are given in Figure 17. Sheep cloning combines the *rename* and *copy* functions.

Transition rules. The transition rules of μENCORE are presented in Figures 20 and 21. Let a denote the map of fields to values in an object and l to denote map of local variables to values in a (possibly blocked) frame. A context reduction semantics decomposes an expression into a reduction context and a redex, and reduces the redex (e.g., [16,29]). A *reduction context* is denoted by an expression E with a single hole denoted by $\bullet$, while an expression without any holes is denoted by e. Filling the hole of a context E with an expression e is denoted by $E[e]$, which represents the expression obtained by replacing the hole of E with

$$
\begin{array}{l}
\mathit{lookup}(v, \varepsilon) = \varepsilon \\
\mathit{lookup}(v, \mathit{obj}\ \mathit{hp}) = \begin{cases} \mathit{obj} & \text{if } \mathit{obj} = v(a) \\ \mathit{lookup}(v, \mathit{hp}) & \text{otherwise} \end{cases} \\
\mathit{lookup}(\texttt{closure}\ \overline{x}' = \overline{v}'\ \texttt{in}\ \lambda(\overline{x} : \overline{T}) \to e : T, \varepsilon) = \varepsilon
\end{array}
$$

$$
\begin{array}{l}
\mathit{rename}(\varepsilon, \mathit{hp}, \sigma) = \sigma \\
\mathit{rename}(v\,\overline{v}, \mathit{hp}, \sigma) = \mathit{rename}(\overline{v}, \mathit{hp}, \sigma)\ \text{if}\ \begin{cases} v \in \mathit{dom}(\sigma), \\ \mathit{lookup}(v, \mathit{hp}) = v(a) \wedge \mathit{active}(v), \text{or} \\ v \text{ is neither an object nor a closure} \end{cases} \\
\mathit{rename}(v\,\overline{v}, \mathit{hp}, \sigma) = \mathit{rename}(\mathit{ran}(a)\,\overline{v}, \mathit{hp}, \sigma[v \mapsto v']) \\
\qquad\qquad \text{if } \mathit{lookup}(v, \mathit{hp}) = v(a) \wedge \neg \mathit{active}(v) \wedge \mathit{fresh}(v') \wedge \neg \mathit{active}(v') \\
\mathit{rename}((\texttt{closure}\ \overline{x}' = \overline{v}'\ \texttt{in}\ \lambda(\overline{x} : \overline{T}) \to e : T)\,\overline{v}, \mathit{hp}, \sigma) = \mathit{rename}(\overline{v}'\,\overline{v}, \mathit{hp}, \sigma)
\end{array}
$$

$$
\begin{array}{l}
\mathit{copy}(\varepsilon, \mathit{hp}, \mathit{transfer}) = \mathit{transfer} \\
\mathit{copy}(v\,\overline{v}, \mathit{hp}, \mathit{transfer}) = \mathit{transfer})\ \text{if}\ \begin{cases} \mathit{lookup}(v, \mathit{hp}) \in \mathit{transfer}, \\ \mathit{lookup}(v, \mathit{hp}) = v(a) \wedge \mathit{active}(v), \mathit{or} \\ v \text{ is neither an object nor a closure} \end{cases} \\
\mathit{copy}(v\,\overline{v}, \mathit{hp}, \mathit{transfer}) = \mathit{copy}(\mathit{ran}(a)\,\overline{v}, \mathit{hp}, \mathit{transfer} \cup \{v(a)\}) \\
\qquad\qquad \text{if } \mathit{lookup}(v, \mathit{hp}) = v(a) \wedge \neg \mathit{active}(v) \\
\mathit{copy}((\texttt{closure}\ \overline{x}' = \overline{v}'\ \texttt{in}\ \lambda(\overline{x} : \overline{T}) \to e : T)\,\overline{v}, \mathit{hp}, \mathit{transfer}) = \mathit{copy}(\overline{v}'\,\overline{v}, \mathit{hp}, \mathit{transfer})
\end{array}
$$

Figure 17. Sheep cloning: deep renaming and copying of passive objects.

$$
\begin{array}{rl}
E ::= & \bullet \mid E; e \mid E? \mid \texttt{if}\ E\ \{\, e_1 \,\}\ \texttt{else}\ \{\, e_2 \,\} \mid x = E \mid \texttt{let}\ \overline{x} = \overline{v}, E, \overline{e}\ \texttt{in}\ \{\, e \,\} \\
& \mid E \diamond m(\overline{e}) \mid v \diamond m(\overline{v}, E, \overline{e}) \mid E \not\downarrow m(\overline{e}) \mid v \not\downarrow m(\overline{v}, E, \overline{e}) \mid \texttt{new}\ C\,(\overline{v}, E, \overline{e}) \mid \texttt{get}\ E \\
& \mid \texttt{spore}\ \overline{x} = \overline{v}, E, \overline{e}\ \texttt{in}\ \lambda(\overline{x} : \overline{T}) \to e : T \mid E(\overline{e}) \mid v(\overline{v}, E, \overline{e}) \mid E \rightsquigarrow e \mid v \rightsquigarrow E
\end{array}
$$

Figure 18. Context reduction semantics of μENCORE: the contexts.

$$
\begin{array}{rl}
\mathit{redexes} ::= & x \mid x.x \mid v \mid v; e \mid v? \mid \texttt{if}\ v\ \{\, e_1 \,\}\ \texttt{else}\ \{\, e_2 \,\} \mid \texttt{await}\ e \mid x = v \mid \texttt{while}\ e_1\ \{\, e_2 \,\} \\
& \mid \texttt{let}\ \overline{x} = \overline{v}\ \texttt{in}\ \{\, e \,\} \mid v \diamond m(\overline{v}) \mid v \not\downarrow m(\overline{v}) \mid \texttt{new}\ C\,(\overline{v}) \mid \texttt{get}\ v \mid \texttt{skip} \mid v \rightsquigarrow v' \\
& \mid \texttt{closure}\ \overline{x}' = \overline{v}'\ \texttt{in}\ (\lambda(\overline{x} : \overline{T}) \to e : T)(\overline{v}) \mid \texttt{spore}\ \overline{x}' = \overline{v}'\ \texttt{in}\ (\lambda(\overline{x} : \overline{T}) \to e : T)
\end{array}
$$

Figure 19. Context reduction semantics of μENCORE: the redexes.

e. In the rules, an expression $E[e]$ consisting of a context E and a redex e is reduced to $E[e']$, possibly with side effects. Here the context E determines the hole where a reduction may occur and e is the redex located in the position of that hole. The contexts of our semantics are given in Figure 18 and the redexes in Figure 19.

Basic rules. Rule SKIP consumes a **skip** in the active task. Rules ASSIGN1 and ASSIGN2 assign a value v to a variable x in the local variables l or in the fields a, respectively. In the rules, the premise $l(\mathit{this}) = o$ looks up the corresponding object. Rule VARIABLE1 reads the value of a local variable or a field of the object executing the frame. Rule VARIABLE2 reads the value of the field of another object in the same local heap. Rules COND1 and COND2 cover the two cases of conditional expression. Rule LET associates a value v to a local variables x and uses it in the expression e. Rule WHILE unfolds the **while** loop

(Skip)
$$g(t(\{l|E[\texttt{skip}]\} \circ sq), hp, q) \rightarrow g(t(\{l|E[()]\} \circ sq), hp, q)$$

(Suspend)
$$g(t(\{l|E[\texttt{suspend}]\} \circ sq), hp, q) \rightarrow g(idle, hp, t(\{l|E[()]\} \circ sq) \circ q)$$

(Sequential)
$$g(t(\{l|E[v; e]\} \circ sq), hp, q) \rightarrow g(t(\{l|E[e]\} \circ sq), hp, q)$$

(Variable1)
$$\frac{l(this) = o \quad o(a) \in hp \quad a \circ l(x) = \langle T, v \rangle}{g(t(\{l|E[x]\} \circ sq), hp, q) \rightarrow g(t(\{l|E[v]\} \circ sq), hp, q)}$$

(Let)
$$\frac{fresh(\overline{y})}{g(t(\{l|E[\texttt{let } \overline{x} = \overline{v} \texttt{ in } \{ e \}]\} \circ sq), hp, q) \rightarrow g(t(\{l[\overline{y} \mapsto \overline{v}]|E[\ e[\overline{x}/\overline{y}]\ ]\} \circ sq), hp, q)}$$

(Variable2)
$$\frac{\{o(a), o'(a')\} \subseteq hp \quad l(this) = o \quad a \circ l(x_1) = \langle T', o' \rangle \quad a'(x_2) = \langle T, v \rangle}{g(t(\{l|E[x_1.x_2]\} \circ sq), hp, q) \rightarrow g(t(\{l|E[v]\} \circ sq), hp, q)}$$

(Activate)
$$\frac{task = select(q, fut) \quad q' = q \setminus \{task\}}{g(idle, hp, q)\ fut \rightarrow g(task, hp, q')\ fut}$$

(Poll-Future1)
$$\frac{fut(f) = v}{g(t(\{l|E[f?]\} \circ sq), hp, q)\ fut \rightarrow g(t(\{l|E[\texttt{true}]\} \circ sq), hp, q)\ fut}$$

(Poll-Future2)
$$\frac{fut(f) = \bot}{g(t(\{l|E[f?]\} \circ sq), hp, q)\ fut \rightarrow g(t(\{l|E[\texttt{false}]\} \circ sq), hp, q)\ fut}$$

(New-Active-Object)
$$\frac{a = atts(C, o) \quad fresh(o) \quad fresh(f) \quad active(C) \quad fut' = fut[f \mapsto \bot] \quad \{l'|e'\} = init(C, f)}{g(t(\{l|E[\texttt{new } C()]\} \circ sq), hp, q)\ fut \rightarrow g(t(\{l|E[o]\} \circ sq), hp, q)\ \ fut' \quad g(t(\{l'[this \mapsto \langle C, o \rangle]|e'\} \circ \text{eos}), \{o(a)\}, \emptyset)}$$

(New-Passive-Object)
$$\frac{fr = \{l'[this \mapsto \langle C, o \rangle]|e'; o\} \quad \{l'|e'\} = init(C) \quad hp' = hp \cup \{o(a)\} \quad \neg active(C) \quad fresh(o) \quad a = atts(C, o)}{g(t(\{l|E[\texttt{new } C()]\} \circ sq), hp, q) \rightarrow g(t(fr \circ \{l|E[\bullet]\} \circ sq), hp', q)}$$

(Assign1)
$$\frac{l(x) = \langle T, v' \rangle \quad l' = l[x \mapsto \langle T, v \rangle]}{g(t(\{l|E[x = v]\} \circ sq), hp, q) \rightarrow g(t(\{l'|E[()]\} \circ sq), hp, q)}$$

(Assign2)
$$\frac{a(x) = \langle T, v' \rangle \quad x \notin dom(l) \quad hp' = hp \cup \{o(a[x \mapsto \langle T, v \rangle])\} \quad l(this) = o}{g(t(\{l|E[x = v]\} \circ sq), hp \cup \{o(a)\}, q) \rightarrow g(t(\{l|E[()]\} \circ sq), hp', q)}$$

(Cond1)
$$g(t(\{l|E[\texttt{if true } \{e_1\} \texttt{ else } \{e_2\}]\} \circ sq), hp, q) \rightarrow g(t(\{l|E[e_1]\} \circ sq), hp, q)$$

(Cond2)
$$g(t(\{l|E[\texttt{if false } \{e_1\} \texttt{ else } \{e_2\}]\} \circ sq), hp, q) \rightarrow g(t(\{l|E[e_2]\} \circ sq), hp, q)$$

(Await)
$$g(t(\{l|E[\texttt{await } e]\} \circ sq), hp, q) \rightarrow g(t(\{l|E[\texttt{if } e? \ \{\texttt{skip}\} \texttt{ else } \{\texttt{suspend}; \texttt{await } e\}]\} \circ sq)\ , hp, q)$$

(While)
$$g(t(\{l|E[\texttt{while } e_1 \ \{ \ e_2 \ \}]\} \circ sq), hp, q) \rightarrow g(t(\{l|E[\texttt{if } e_1 \ \{e_2; \texttt{while } e_1 \ \{ \ e_2 \ \}\} \texttt{ else } \{\texttt{skip}\}]\} \circ sq), hp, q)$$

Figure 20. Semantics for μEncore (1).

into a conditional. Rule Sequential discards the value v in an expression of the form $v; e$ and continues the evaluation the expression e.

Suspension and activation. Rule Suspend enables cooperative scheduling and moves the active task to the queue q, making the active task *idle*. Rule Await unfolds into a conditional. Rules Poll-Future1 and Poll-Future2

$$\frac{fut' = fut[f \mapsto \bot] \quad abind(m, o, \overline{v}, f) = fr \quad o \in dom(hp) \quad fresh(f) \quad q' = t(fr \circ \text{eos}) \circ q}{g(t(\{l|E[o\,!\,m(\overline{v})]\} \circ sq), hp, q)\ fut \rightarrow g(t(\{l|E[f]\} \circ sq), hp, q')\ fut'}\ (\textsc{Local-Async-Call})$$

$$\frac{o(a) \in hp \quad q' = \{t(fr \circ \text{eos})\} \cup q \quad fresh(s) \quad abind(m, o, \overline{v}, f) = fr}{g(active, hp, q)\ m(o, \overline{v}, hp', f) \rightarrow g(active, hp \cup hp', q')}\ (\textsc{Remote-Bind-Mtd})$$

$$\frac{o \notin dom(hp) \quad fresh(f) \quad \overline{v}' = \overline{v}\,\sigma \quad rename(\overline{v}, hp, \varepsilon) = \sigma \quad fut' = fut[f \mapsto \bot] \quad copy(\overline{v}', hp\,\sigma, \varepsilon) = hp'}{g(t(\{l|E[o\,!\,m(\overline{v})]\} \circ sq), hp, q)\ fut \rightarrow g(t(\{l|E[f]\} \circ sq), hp, q)\ m(o, \overline{v}', hp', f)\ fut'}\ (\textsc{Remote-Async-Call})$$

$$\frac{f = l(destiny) \quad fut' = fut[f \mapsto v]}{g(t(\{l|v\} \circ \text{eos}), hp, q)\ fut \rightarrow g(idle, hp, q)\ fut'}\ (\textsc{Async-Return})$$

$$\frac{fut(f) = v \quad v \neq \bot}{g(t(\{l|E[\texttt{get}\ f]\} \circ sq), hp, q)\ fut \rightarrow g(t(\{l|E[v]\} \circ sq), hp, q)\ fut}\ (\textsc{Read-Fut})$$

$$\frac{o \in dom(hp) \quad bind(m, o, \overline{v}) = fr}{g(t(\{l|E[o \diamond m(\overline{v})]\} \circ sq), hp, q) \rightarrow g(t(fr \circ \{l|E[\bullet]\} \circ sq), hp, q)}\ (\textsc{Sync-Call})$$

$$g(t(\{l|v\} \circ \{l'|E[\bullet]\} \circ sq), hp, q) \rightarrow g(t(\{l'|E[v]\} \circ sq), hp, q)\ (\textsc{Sync-Return})$$

$$\frac{\overline{v}' = \overline{v}\sigma \quad rename(\overline{v}, hp, \varepsilon) = \sigma \quad copy(\overline{v}', hp\,\sigma, \varepsilon) = hp'}{g(t(\{l \mid E[(\texttt{spore}\ \overline{x}' = \overline{v}\ \texttt{in}\ \lambda(\overline{x} : \overline{T}) \rightarrow e : T)]\} \circ sq), hp, q) \rightarrow g(t(\{l \mid E[(\texttt{closure}\ \overline{x}' = \overline{v}'\ \texttt{in}\ \lambda(\overline{x} : \overline{T}) \rightarrow e : T)]\} \circ sq), hp \cup hp', q)}\ (\textsc{Create-Closure})$$

$$g(t(\{l \mid E[(\texttt{closure}\ \overline{x}' = \overline{v}'\ \texttt{in}\ \lambda(\overline{x} : \overline{T}) \rightarrow e : T)(\overline{v})]\} \circ sq), hp, q) \rightarrow g(t(\{l \mid E[\texttt{let}\ \overline{x}', \overline{x} = \overline{v}', \overline{v}\ \texttt{in}\ \{\ e\ \}]\} \circ sq), hp, q)\ (\textsc{Lambda-App})$$

$$\frac{\overline{v}' = \overline{v}\sigma \quad hp' = copy(\overline{v}', hp\,\sigma, \varepsilon) \cup \{o(\varepsilon)\} \quad rename(\overline{v}, hp, \varepsilon) = \sigma \quad v_1 = \texttt{closure}\ \overline{x} = \overline{v}\ \texttt{in}\ \lambda(x : T) \rightarrow e : T' \quad fresh(o) \quad fresh(f') \quad v_2 = \texttt{closure}\ \overline{x} = \overline{v}'\ \texttt{in}\ \lambda(x : T) \rightarrow e : T' \quad l' = [this \mapsto \langle \mathsf{Closure}, o\rangle, destiny \mapsto \langle \mathsf{Fut}\ T', f'\rangle] \quad fut' = fut[f' \mapsto \bot]}{g(t(\{l|E[f \rightsquigarrow v_1]\} \circ sq), hp, q)\ fut \rightarrow g(t(\{l|E[f']\} \circ sq), hp, q)\ g(t(\{l'|v_2\ (\texttt{get}\ f)\} \circ \text{eos}), \{hp'\}, \emptyset)\ fut'}\ (\textsc{New-Chained-Object})$$

Figure 21. Semantics for μENCORE (2).

test whether the future f has been resolved. If f is not resolved, the active task suspends. Otherwise, the **`await`** expression is reduced to a **`skip`**. Rule ACTIVATE schedules a task from the task queue q by means of the *select* function. Since the schedulability of a task may depend on a future, *select* uses the map of futures *fut*.

Asynchronous method calls. Rule REMOTE-ASYNC-CALL sends an invocation message to an object o, with a return address f and sheep copied actual parameter values. The cloned objects hp' are transferred with the method invocation. (Notation $\overline{v}\,\sigma$ and $hp\,\sigma$ denote the recursive application of the substitution σ to $\overline{v}$ and hp, respectively). The identifier of the new future is added to *fut* with

a $\perp$ value indicating that it has not been fulfilled. In rule Remote-Bind-Mtd, the function $abind(m, o, \bar{v}, f)$ binds a method call in the class of the callee o. This results in a new task in the queue of the active object o. In the frame fr, the local variable *this* is bound to o and *destiny* is bound to f. The heap hp' transferred with the message extends the heap hp of the active object. Rule Local-Async-Call puts a new task with a single frame fr on the queue q. As before, a new future f is created and associated to the variable *destiny* in fr, the identifier of the new future is added to *fut* with a $\perp$ value. Rule Async-Return places the final value of a task into the associated future, making the active task idle. Rule Read-Fut dereferences a future f from the maps of futures *fut*.

Synchronous method calls. In rule Sync-Call, method m is called on a local object o, with actual parameters $\overline{v}$. The function $bind(m, o', \overline{v}')$ binds the call in the class of o, resulting in a frame fr. The new frame extends the stack, and the previously active frame becomes blocked. In rule Sync-Return, the active frame only contains a single value. The frame is popped form the stack, and the value is passed to the blocked frame below which becomes active.

Object creation. Rule New-Active-Object creates a an active object with a unique identifier o' and a new local heap. The fields of o' are given default values by $atts(C, o')$. The active task of the new active object is the constructor $init(C, f)$, where the local variable *this* binds to $\langle C, o' \rangle$ and *destiny* binds to $\langle \mathsf{Fut}\ C, f \rangle$.[11] Passive object are created in a similar way, except that the class constructor is executed by the calling thread (cf. Rule New-Passive-Object).

Closures. In rule Create-Closure, a closure is created from a spore by sheep cloning any references to the passive objects of enclosing active object. Note that values inside closures are also sheep copied, even if this has already been done when they were created, to ensure that if the closure is passed out of the active object, it is passed with a fresh sheep clone of its passive objects. Rule Lambda-App reduces a closure to a let-expression when it is applied to values.

Future chaining. Future chaining creates a new dummy active object in which the closure can execute in parallel with the current active object. The closure blocks waiting for the value of it needs from f to begin execution, and will return its own value to another future f'.

12.4 Run-time Typing

Assume a typing context $CT(C)$ that maps fields of each class C to their declared types. The class table also includes a class $\mathsf{Closure}$ such that $CT(\mathsf{Closure}) = \varepsilon$. The run-time type system (Figure 22) facilitates the type preservation proof for μEncore.

Lemma 1. *If a program $\overline{IF}\ \overline{CL}\ e$ is well-typed, then there is a Γ such that the initial run-time state of this program is well-typed: $\Gamma \vdash s(\{this \mapsto \langle \mathsf{Closure}, o \rangle | e\} \circ eos, o(\varepsilon), \emptyset)$ $\mathtt{ok}$.*

[11] In Encore the constructor does not run asynchronously.

$$\text{(RT-Empty)}\ \frac{}{\Gamma \vdash \varepsilon\ \texttt{ok}} \qquad \text{(RT-Idle)}\ \frac{}{\Gamma \vdash \mathit{idle}\ \texttt{ok}} \qquad \text{(RT-Oid)}\ \frac{}{\Gamma \vdash v : \Gamma(v)} \qquad \text{(RT-Queue1)}\ \frac{}{\Gamma \vdash \emptyset\ \texttt{ok}}$$

$$\text{(RT-Poll)}\ \frac{\Gamma \vdash e : \mathsf{Fut}\langle T\rangle}{\Gamma \vdash e? : \mathsf{Bool}} \qquad \text{(RT-Fut1)}\ \frac{\Gamma \vdash \mathit{fut}\ \texttt{ok}}{\Gamma \vdash \mathit{fut}[f \mapsto \bot]\ \texttt{ok}} \qquad \text{(RT-Fut2)}\ \frac{\Gamma \vdash v : T \quad \Gamma \vdash \mathit{fut}\ \texttt{ok} \quad \Gamma(f) = \mathsf{Fut}\ T}{\Gamma \vdash \mathit{fut}[f \mapsto v]\ \texttt{ok}} \qquad \text{(RT-Task)}\ \frac{\Gamma \vdash fr \circ sq : \mathsf{Unit}}{\Gamma \vdash s(fr \circ sq)\ \texttt{ok}}$$

$$\text{(RT-Stack2)}\ \frac{\Gamma \vdash \{l|e\} : T \quad l(\mathit{destiny}) = \langle \mathsf{Fut}\langle T\rangle, f\rangle}{\Gamma \vdash \{l|e\} \circ \mathit{eos} : T} \qquad \text{(RT-Queue2)}\ \frac{\Gamma \vdash q_1\ \texttt{ok} \quad \Gamma \vdash q_2\ \texttt{ok}}{\Gamma \vdash q_1 \cup q_2\ \texttt{ok}} \qquad \text{(RT-Subst1)}\ \frac{\Gamma(x) = T \quad \Gamma \vdash v : T}{\Gamma \vdash x \mapsto \langle T, v\rangle\ \texttt{ok}} \qquad \text{(RT-Subst2)}\ \frac{\Gamma \vdash \sigma_1\ \texttt{ok} \quad \Gamma \vdash \sigma_2\ \texttt{ok}}{\Gamma \vdash \sigma_1 \circ \sigma_2\ \texttt{ok}}$$

$$\text{(RT-Frame)}\ \frac{\sigma(\mathit{this}) = \langle C, o\rangle \quad \Gamma' \vdash \sigma\ \texttt{ok} \quad \Gamma' \vdash e : T \quad \sigma = \overline{x} \mapsto \langle \overline{T}, \overline{v}\rangle \quad \Gamma' = \Gamma \circ CT(C) \circ \overline{x} \mapsto \overline{T}}{\Gamma \vdash \{\sigma|e\} : T} \qquad \text{(RT-Msg)}\ \frac{\Gamma \vdash o : T' \quad \Gamma \vdash \overline{v} : \overline{T} \quad \mathit{typeOf}(T', m) = \overline{T} \to T \quad \Gamma \vdash f : \mathsf{Fut}\ T \quad \Gamma \vdash \mathit{hp}\ \texttt{ok}}{\Gamma \vdash m(o, \overline{v}, \mathit{hp}, f)\ \texttt{ok}}$$

$$\text{(RT-Obj)}\ \frac{\Gamma(o) = C \quad \Gamma \circ CT(C) \vdash \sigma\ \texttt{ok}}{\Gamma \vdash o(\sigma)\ \texttt{ok}} \qquad \text{(RT-Heap)}\ \frac{\Gamma \vdash \mathit{hp}_1\ \texttt{ok} \quad \Gamma \vdash \mathit{hp}_2\ \texttt{ok}}{\Gamma \vdash \mathit{hp}_1 \cup \mathit{hp}_2\ \texttt{ok}} \qquad \text{(RT-Conf)}\ \frac{\Gamma \vdash \mathit{cn}_1\ \texttt{ok} \quad \Gamma \vdash \mathit{cn}_2\ \texttt{ok}}{\Gamma \vdash \mathit{cn}_1\ \mathit{cn}_2\ \texttt{ok}} \qquad \text{(RT-Group)}\ \frac{\Gamma \vdash \mathit{active} : T \quad \Gamma \vdash \mathit{hp}\ \texttt{ok} \quad \Gamma \vdash q\ \texttt{ok}}{\Gamma \vdash g(\mathit{active}, \mathit{hp}, q)\ \texttt{ok}}$$

$$\text{(RT-Closure)}\ \frac{\Gamma \vdash (\texttt{spore}\ \overline{x} = \overline{v}\ \texttt{in}\ \lambda(\overline{x}' : \overline{T}) \to e : T) : \overline{T} \Rightarrow T}{\Gamma \vdash (\texttt{closure}\ \overline{x} = \overline{v}\ \texttt{in}\ \lambda(\overline{x}' : \overline{T}) \to e : T) : \overline{T} \Rightarrow T}$$

Figure 22. Type system for μEncore run-time states.

Lemma 2 (Sheep lemma). *Assume that $\Gamma \vdash hp$ **ok** and let σ be a substitution such that $dom(\sigma) \subseteq dom(\Gamma)$, $ran(\sigma) \cap dom(\Gamma) = \emptyset$. Let $\Gamma' = \{y \mapsto T | \sigma(x) = y \wedge \Gamma(x) = T\}$ Then $\Gamma \circ \Gamma' \vdash hp\,\sigma$ **ok**.*

Lemma 3 (Type preservation). *If $\Gamma \vdash cn$ **ok** and $cn \to cn'$ then there exists a Γ' such that $\Gamma \subseteq \Gamma'$ and $\Gamma \vdash cn'$ **ok***

Theorem 1. *Let P be a program in μEncore, with an initial state cn. If $\Gamma \vdash P$ **ok** and $cn \to cn'$, there is a typing environment Γ' such that $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash cn'$ **ok**.*

Proof. Follows directly from Lemmas 1–3.

13 Related Work

Encore is based on the concurrency model provided by active objects and actor-based computation, where software units with encapsulated thread of control communicate asynchronously. Languages based on the actor model [1, 2] take asynchronous messages as the communication primitive and focus on loosely

coupled processes with less synchronisation. This makes actor languages conceptually attractive for parallel programming programming. Rather than the pure asynchronous message passing model of actor systems, active objects adopts method calls as asynchronous messages combined with futures to deliver results. Futures were devised as a means to reduce latency by synchronising at the latest possible time. Futures were discovered by Baker and Hewitt in the 70s [5], and rediscovered after around 10 years and introduced in languages such as ABCL [41, 42], Argus [25], ConcurrentSmalltalk [40], and MultiLisp [19] and later in Alice [34], Oz-Mozart [37], Concurrent ML [33], C++ [24] and Java [38], often as libraries. Nowadays, active object and actor-based concurrency is increasingly attracting attention due to its intuitive and compositional nature, which can lead to good scalability in a parallel setting. Modern example languages or frameworks include Erlang [4], ProActive [7], Scala Actors [18], Kilim [35], Creol [22], ABS [21], Akka [36], Habanero-Java [9], among others.

Encore has a clear distinction between active and passive objects, such that passive objects as a default are only locally accessible. This is ensured in μEncore by means of sheep cloning [11] and paves the way for capability type systems for sharing, as first investigated in Joëlle [14]. Encore further features spores, originally proposed for Scala [27]. Although spores in Encore need not be pure, they are combined with sheep cloning to preserve race-free execution as a default. Future versions of spores in Encore will utilise capabilities for more fine-grained control.

14 Conclusion

Programming parallel computers is hard, but as all future computers will be parallel computers doing so will be a necessary skill of all programmers. New programming models and languages are required to support programmers in writing applications that are safe and exploit the available parallel computing resources. Encore aims to answer this challenge by provided active-object based parallelism combined with additional mechanisms such as parallel combinators and capabilities for safely expressing other forms of parallelism. This paper gave an overview of Encore, including the semantics of its core, along with a number of examples showing how to use the language.

Work on Encore has really only just begun. In the future we will be implementing and improving language constructs for expressing different kinds of parallelism and for controlling sharing, data layout and other deployment related concerns. We will continue improving the run-time system, developing libraries and tool support, and exploring case studies. In the near future, we plan to convert the compiler to open source. When this happens—or even beforehand, if you are keen—, you are more than welcome to contribute to the development of Encore.

A Code for Preferential Attachment

```
class Worker
  id : int
  k : int
  owners_map : Map<int,Worker>
  n_size : int
  n_start : int
  outstanding : int
  edges : Set<int>
  home : Main
  connections : Array<int>

  def init(id:int, k:int, cs:Array<int>, map:Map<int,Worker>, m:Main) : void {
    this.id = id;
    this.k = k;
    this.owners_map = map;
    this.n_size = map.range / k;
    this.n_start = this.n_size * id;
    this.edges = new Set<int>(k);
    this.home = m;
    this.connections = cs;
  }

  def check_done() : void
    if this.outstanding == 0 then this.home ! done(this.id)

  def start() : void {
    let
      start = this.n_start
    in
      repeat i <- this.n_size
        this.add_node(start + i);

    this.check_done()
  }

  def add_node(n:int) : void {
    this.edges.reset();

    let
      start = n * this.k
    in
      repeat i <- this.k
        this.add_edge(n, start + i)
  }

  def add_edge(n:int, target_i:int) : void
    let
      coin_flip = this.random(0, 2)
      from_i = this.random(0, n * this.k)
      c = if coin_flip == 0 then (from_i % n) + 1
                            else this.connections.read(from_i)
    in
```

```
53          if c < 0 then this.add_edge(n, target_i)
54     else if c == 0 then this.add_remote_edge(target_i, from_i);
55     else this.add(n, target_i, c);
56
57  def add_remote_edge(local_i:int, remote_i:int) : void {
58    this.outstanding = this.outstanding + 1;
59    this.remote_actor(remote_i) ! read_location(remote_i, local_i, this);
60  }
61
62  def read_location(local_i:int, remote_i:int, from:Worker) : void
63    let
64      c = this.connections.read(local_i)
65    in
66          if c > 0 then from ! add_edge_from_remote(remote_i, c, this)
67     else if c == 0 then this ! read_location(local_i, remote_i, from)
68
69  def redo(i:int) : void {
70    this.add_edge(i / this.k, i);
71    this.outstanding = this.outstanding - 1;
72  }
73
74  def add_edge_from_remote(i:int, c:int, from:Worker) : void {
75    this.outstanding = this.outstanding - 1;
76    this.edges.reset();
77
78    let start_i = i - (i % this.k) in repeat j <- this.k
79      let
80        v = this.connections.read(start_i + j)
81      in
82        if not this.edges.add(v) then unless v == 0 then
83          this.add_edge(i / this.k, start_i + j);
84
85    if this.edges.add(c) then this.add_local_edge(i, c)
86                         else this.add_edge(i / this.k, i);
87
88    this.check_done();
89  }
90
91  def add_local_edge(i:int, c:int) : void
92    this.connections.write(i, c)
93
94  def remote_actor(i:int) : Worker
95    this.owners_map.lookup(i / this.k)
96
97  def add(n:int, i:int, c:int) : void
98        if c < 0 then this.add_local_edge(i, c)
99    else if this.edges.add(c) then this.add_local_edge(i, c)
100   else this.add_edge(n, i)
101
102 def random(a:int,b:int) : int
103   embed int
104     (random() % b) + a;
105   end
106
```

```
  def generate_clique() : void {
    repeat n <- this.k
      repeat k <- this.k
        let i = k + (n * this.k) in
          this.add_local_edge(i, if k < n then k+1 else (0-1));

    repeat n <- (this.n_size - this.k)
      this.add_node(this.k + n);

    this.home ! done(this.id);
  }

class Main
  workers:int

  def done(id:int) : void {
    this.workers = this.workers - 1;
    if this.workers == 0 then print("Done!\n");
  }

  def fix_mod_assert(n:int, k:int, w:int) : int
    (n * k) % w

  def main() : void
    let
      n = 1000000
      k = 32
      workers = 32
      array = new Array<int>(n * k);
      owners_map = new Map<int,Worker>(n * k, workers);
    in
      {
        embed void
          if (argc > 3) {
            #{n} = atoi(argv[1]);
            #{k} = atoi(argv[2]);
            #{workers} = atoi(argv[3]);
          }
        end;
        -- To avoid stupid rounding errors when splitting
        -- the array across actors
        assertTrue(this.fix_mod_assert(n, k, workers) == 0);
        assertTrue(n > k * workers);
        repeat i <- workers
          let
            a = new Worker(i, k, array, owners_map, this)
          in
            {
              this.owners_map.add(a);

              if i == 0 then a ! generate_clique()
                        else a ! start();
            }
      }
```

References

1. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1986.
2. G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, Jan. 1997.
3. R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47–97, Jan 2002.
4. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
5. H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, Aug. 1977.
6. A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
7. D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous sequential processes. *Information and Computation*, 207(4):459–495, 2009.
8. E. Castegren and T. Wrigstad. Capable: Capabilities for scalability. IWACO 2014, 2014.
9. V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. In C. W. Probst and C. Wimmer, editors, *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, Kongens Lyngby, Denmark, August 24-26, 2011*, pages 51–61. ACM, 2011.
10. D. Clarke. *Object ownership and containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Australia, 2002.
11. D. Clarke, J. Noble, and T. Wrigstad, editors. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*. Springer, 2013.
12. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer, 2003.
13. D. Clarke, T. Wrigstad, J. Östlund, and E. Johnsen. Minimal ownership for active objects. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 139–154. Springer Berlin / Heidelberg, 2008.
14. D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal ownership for active objects. In G. Ramalingam, editor, *Proc. 6th Asian Symposium on Programming Languages and Systems (APLAS'08)*, volume 5356 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2008.
15. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.
16. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.
17. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification, (3rd Edition)*. Addison-Wesley Professional, 2005.
18. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
19. R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

20. L. Henrio, F. Huet, and Z. István. Multi-threaded active objects. In R. D. Nicola and C. Julien, editors, *Coordination Models and Languages, 15th International Conference, COORDINATION 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, volume 7890 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2013.
21. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.
22. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
23. D. Kitchin, A. Quark, W. R. Cook, and J. Misra. The orc programming language. In D. Lee, A. Lopes, and A. Poetzsch-Heffter, editors, *Formal Techniques for Distributed Systems, Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5522 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2009.
24. R. G. Lavender and D. C. Schmidt. Pattern languages of program design 2. chapter Active Object: An Object Behavioral Pattern for Concurrent Programming, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
25. B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proceedings of the SIGPLAN Conference on Programming Lanugage Design and Implementation (PLDI'88)*, pages 260–267, Atlanta, GE, USA, June 1988. ACM.
26. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
27. H. Miller, P. Haller, and M. Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In R. Jones, editor, *Proc. 28th European Conference on Object-Oriented Programming (ECOOP'14)*, Lecture Notes in Computer Science, pages 308–333. Springer, 2014.
28. M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.
29. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
30. J. Noble, D. G. Clarke, and J. Potter. Object ownership for dynamic alias protection. In *TOOLS Pacific 1999: 32nd International Conference on Technology of Object-Oriented Languages and Systems, 22-25 November 1999, Melbourne, Australia*, pages 176–187. IEEE Computer Society, 1999.
31. S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.
32. A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A generic parallel collection framework. In E. Jeannot, R. Namyst, and J. Roman, editors, *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II*, volume 6853 of *Lecture Notes in Computer Science*, pages 136–147. Springer, 2011.
33. J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

34. A. Rossberg, D. L. Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, ISBN 1-84150144-1, Munich, Germany, Feb. 2006.
35. S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In J. Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*, pages 104–128. Springer, 2008.
36. The Akka Project. Akka, 2015. `http://akka.io/`.
37. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Mar. 2004.
38. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. Object oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 439–453, New York, NY, USA, 2005. ACM Press.
39. T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for Java. In *ECOOP 2009 – Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 445–469. Springer Berlin / Heidelberg, 2009.
40. Y. Yokote and M. Tokoro. Concurrent programming in ConcurrentSmalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. The MIT Press, Cambridge, Mass., 1987.
41. A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. Series in Computer Systems. The MIT Press, 1990.
42. A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86). Sigplan Notices*, 21(11):258–268, Nov. 1986.