Exceptional Actors
Implementing Exception Handling for Encore

Sahand Shamal Taher
This page intentionally left blank
Abstract

Exceptional Actors: Implementing Exception Handling for Encore

Sahand Shamal Taher

Encore is an object-oriented programming language which uses the actor model as its concurrency model, and is specifically aimed at implementing concurrent and parallel systems. Communication between actors is done via asynchronous method calls, which store their results in futures, placeholder objects for later storing the result of an computation. Encore currently lacks an exception handling mechanism, which is an important part of programming languages, and helps programmers create more fault tolerant and robust software. This thesis presents an exception handling implementation for Encore, in which futures are used to propagate errors between actors. The implementation uses a modified version of the open-source library Exceptions4C as a basis, and enables basic exception handling through try-catch-finally expressions. It uses type-based matching that also considers subtyping, and allows the programmer to define and throw custom exception types. A pull-model is used to enable inter-process communication of exceptions, where receiving actors can choose to handle exceptions at any point by consuming associated futures. The implementation is a good first step, but there are Encore features such as streams which it does not yet support. Furthermore, it brings an overhead to programs, which can be reduced by redesigning parts of the exception handling model.
This page intentionally left blank
# Contents

1 Introduction 1

2 Background 3
2.1 ENCORE .................................................. 3
   2.1.1 Message Passing and Futures .................. 3
   2.1.2 Cooperative Scheduling Primitives .......... 4
   2.1.3 Threads and Scheduling ....................... 5
2.2 Exceptions .............................................. 6
   2.2.1 Exception Handling in C ..................... 7
   2.2.2 The EXCEPTIONS4C Library .................. 8
2.3 Related Work ........................................... 10

3 Exception Handling in Encore 17
3.1 Synchronous Exceptions ......................... 17
3.2 Asynchronous Exceptions ....................... 19

4 Implementation 21
4.1 Language Constructs ............................... 21
4.2 Exception Objects .................................. 23
4.3 Communicating Exceptions Between Actors .... 25
4.4 Protecting the Runtime and Future Fulfils .... 25
   4.4.1 ENCORE’s Dispatch Mechanism ............... 26
   4.4.2 Modifying the Dispatch Mechanism .......... 26
4.5 Scope of Exception Contexts ..................... 27

5 Evaluation 29
5.1 Expressiveness ....................................... 29
   5.1.1 Synchronous exceptions ..................... 29
   5.1.2 Asynchronous Exceptions ................... 31
5.2 Performance ........................................... 32
   5.2.1 Runtime Overhead Experiments ............... 32
   5.2.2 Scalability Experiments ..................... 33
   5.2.3 Results ........................................... 33
5.3 Limitations ............................................ 36
1. Introduction

Exception handling is an important feature of programming languages and helps programmers create more fault tolerant and robust software. As distributed processing has gained increased attention, new concurrency models and exception handling models have emerged. The actor model is a concurrency model that is gaining increased attention [1, 4, 10, 14, 19–21, 36], and has been implemented for languages such as JAVA [18] and SCALA [30], and in the .NET framework [7]. Some languages, such as DART [8] and ERLANG [3, 4], are even built with the actor model as the default concurrency model.

ENCORE, developed by the programming language group at Uppsala University, is an object-oriented programming language which uses the actor model as its concurrency model, and is specifically aimed at implementing concurrent and parallel systems [9]. Communication between actors is done via asynchronous method calls, which store their results in futures, placeholder objects for later storing the result of a computation [5, 24]. The ENCORE compiler is written in HASKELL and translates ENCORE code to C code, which links with a runtime system written in C.

Goal The ENCORE language currently lacks an exception handling mechanism, providing the programmer no help in recovering from errors. To solve this, ENCORE needs an exception handling mechanism which fits in with the concurrency model of ENCORE and preserves the semantics of existing language while using a reasonable amount of resources. As a starting point for ENCORE’s exception handling, the ENCORE team has suggested a JAVA-inspired model to handle exceptions in synchronous settings, and to use futures to communicate exceptions asynchronously, inspired by the paper Fault in the Future [22].

The goal of this thesis is to implement an exception handling mechanism for ENCORE, based on the suggestions of the ENCORE team. A JAVA-inspired model for exception handling in synchronous settings will be provided, and futures be utilized to communicate exceptions in asynchronous settings.

Contributions Concretely, this thesis makes the following contributions to the design and implementation of the ENCORE programming language:

1. It extends ENCORE with a new mechanism for raising and handling exceptions using a `throw` and `try-catch-finally` mechanism. The imple-
mented **try-catch-finally** are expressions which return values.

2. It extends ENCORE with a family of exceptions which can be extended by a programmer. This allows distinguishing between different types of exceptions.

3. It extends ENCORE with an exception, representing an instance of a thrown exception. The exception objects give the programmer access to exception data, and a method for rethrowing the exception.

4. It provides exception handling in asynchronous settings by extending ENCORE’s futures and the `get` mechanism. The futures are modified to store exceptions, and the `get` operation to rethrow exceptions in futures. The context swapping mechanism is extended to support exception handling contexts, which contain message level exception handling information.

5. It provides ENCORE with support to deal with exceptions in functional settings. An **Either**-object encapsulates a value, and either contains a normal value or an exception. Programs can distinguish between the normal and exceptional value through pattern matching.

In Chapter 2 relevant background information is presented, providing a basic understanding of ENCORE, exception handling and exception handling in existing languages. The exception handling model implemented in this work is presented in Chapter 3, where new language constructs are introduced, and the use of futures during exception handling is explained. Implementation details for exception handling model are outlined in Chapter 4, where code generation for new language constructs is shown, an exception class is introduced, and changes to the runtime and context swapping is explained. In Chapter 5, the exception handling implementation is evaluated in terms of expressiveness, performance, and limitations. A conclusion of the work is given in Chapter 6, along with a discussion about the results and future work.
2. Background

This chapter introduces ENCORE and its concurrency model (Section 2.1) and basic exception handling (Section 2.2). Basic exception handling in C is discussed in Section 2.2.1, and a C library for exception handling used in this work presented in Section 2.2.2. Exception handling in related programming languages and works is discussed in (Section 2.3).

2.1 Encore

ENCORE is an object-oriented programming language which uses the actor model as its concurrency model, and is specifically aimed at implementing concurrent and parallel systems [9]. Like regular objects, ENCORE’s actors have encapsulated fields and methods that operate on those fields. Unlike objects, however, a thread of control is associated with each actor, which is the only thread able to access the actor’s fields [9]. Communication between actors is done via asynchronous method calls, which store their results in futures (Section 2.1.1), placeholders for later storing the result of a method call [5, 24]. The ENCORE compiler is written in HASKELL and translates ENCORE code to C code, which links with a runtime system written in C. ENCORE’s runtime runs on top of the runtime of the PONY programming language [25], which is an actor-based, object-oriented programming language. The PONY runtime provides an implementation of the actor model and garbage collection, which ENCORE builds on.

The rest of this section details different parts of ENCORE’s concurrency model that are relevant when implementing exception handling for ENCORE. Message passing and futures, which provide a means for asynchronous communication, are explained in Section 2.1.1, and cooperative scheduling primitives and their affect on actors in Section 2.1.2. Section 2.1.3 discusses ENCORE’s threads and how they are scheduled.

2.1.1 Message Passing and Futures

Communication between actors is done via asynchronous method calls, or message sends, by using the “!” operator. Asynchronous method calls immediately return, optionally with a future, which is used for synchronization and make
blocking operations possible (Figure 2.2). Upon finishing a method, the called actor fulfills or resolves a future with the computed result, which the caller can later retrieve from the future. Each actor has its own inbox, where messages to the actor are placed, each message corresponding to running some method. Actors can only execute one message at a time, though several messages may be running concurrently (explained in Section 2.1.3).

![Figure 2.1: The message queue and currently executing message.](image)

### 2.1.2 Cooperative Scheduling Primitives

The constructs `await`, `get` and `suspend` are cooperative scheduling primitives. The `await` and `get` mechanisms are used for synchronization, whereas `suspend` only affects scheduling. The `get` operation can be used to at any time retrieve a result in a future. This blocks the actor until the future is resolved and then returns the computed value.

``` scala
active class MyActor
  def foo() : unit
  var fut = otherActor ! doThings()
  get(fut)
end
end
```

Listing 2.1: Using the `get` operation on futures without blocking the underlying thread. The actor is blocked if the future is not resolved, and does not execute messages in the meantime if so.

The `await` operation can be used on a future to block the currently executing message, and not the entire actor, until a future is resolved. This allows the actor to process other messages in the meantime.

``` scala
active class MyActor
  def foo() : unit
  var fut = otherActor ! doThings()
  await(fut)
  -- Blocking the executing message. The actor may
  -- execute other messages in the meantime.
```

---

4
get(fut)

-- The actor is not blocked, since the use of 'await'
-- guarantees that the future is fulfilled.
end
end

Listing 2.2: Using the `await` operation to block an executing message

Figure 2.2: Using futures to allow blocking on an asynchronous operation. A message is sent to the account actor that then stores the result in a future. The caller is blocked if it tries to get the result before the future has been resolved.

The `suspend` command suspends an actor indefinitely. After `suspend`, either another message is executed or the suspended message is immediately resumes.

active class MyActor
def foo () : unit
var fut = otherActor ! doThings ()
this.suspend () -- Suspend actor indefinitely
get(fut)
end
end

Listing 2.3: Using the `get` operation on futures

2.1.3 Threads and Scheduling

ENCORE is implemented on top of PTHREADS [27], and each ENCORE program is by default assigned one pthread per CPU core to run the user program. Each pthread can run multiple ENCORE threads or run messages for many actors, and actors can migrate between pthreads. Pthread local actor data is stored as an actor runs, which the scheduler swaps once the actor is finished or a cooperative scheduling primitive is used. Although actors only execute one message at a time, several messages may be running concurrently, in the sense that a message may be awaiting on a future or be suspended, while another of the actor’s messages is executing. Also, if an actor migrates after an `await`,
get or suspend operation, messages can be resumed on a different pthread (Figure 2.3).

Figure 2.3: Interleaved execution of messages. After an await or suspend in Message 2, actor B executes other messages. Before resuming Message 2, the actor will have executed a message on a different thread as well as one on the same thread.

2.2 Exceptions

There are different kinds of errors that can occur in the context of computer programs. A program may compile and execute successfully, but behave in an unexpected way due to a bug, a logic error. On the other hand, the program may not at all compile due to incorrect syntax or failed type checking, a compile time error. An error that occurs while the program is running is called a runtime error, and can crash the program unless handled.

Exceptions can be thought of as unexpected or unusual events during the flow of a program, and are in this work synonymous with runtime errors. A common idiom for exceptions is that they can be raised somewhere in the program, which causes the program to transfer control to a different point, where the exception can be handled and recovered from [17, 33]. By enclosing code in exception handling expressions, the programmer can define error handlers for different kinds of errors. When an error occurs and an exception is raised, the program transfers control to the nearest enclosing corresponding handler, while skipping handlers designated to other kinds of exceptions (Figure 2.4). If no handler is designated to the exception, the exception is said to be uncaught, typically leading to the program exiting.
To allow programs to release resources in the event of errors, many exception handlers provide finalization or clean-up clauses, which allow some code to be executed whether or not an exception occurred in the exception handling expression. Using common terminology, an exception is raised when it is thrown, code that is enclosed in exception handling expression is put inside a `try` block, `catch` blocks define exception handlers, and `finally` blocks define finalizers.

Throughout this thesis, there is a distinction made between exception handling in synchronous and asynchronous settings. Asynchronous exception handling involves passing exceptions between different threads of control, through futures or other mechanisms. To save time, an open source C library for exception handling, presented in Section 2.2.2, is used in the implementation. To provide an understanding of how exception handling is implemented in the exception handling library, the next section first introduces basic exception handling in C.

### 2.2.1 Exception Handling in C

Basic exception handling in C can be implemented with help of the `setjmp.h` library [11, 33], which provides functions for saving and restoring stack contexts. The `setjmp` function saves the current context in a buffer (`jmp_buf`), and when called directly returns 0. The `longjmp` function restores the last saved context and jumps to the position where the context was saved. After a `longjmp` subsequent calls of `setjmp` return a nonzero value assigned by the last `longjmp`.

An example of basic `try-catch` functionality is shown in Listing 2.4, where a conditional statement determines the program status. The first branch of the conditional represents the `try` block, and subsequent branches exception handlers (catch blocks). An initial `setjmp` prepares the program to run the `try` code. After the `throw`, a jump is made to the `setjmp` statement and the conditional is re-entered, this time ending up in an exceptional branch.

```c
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

int main() {
```
int returncode = setjmp(buf); /* longjmp will jump here */
switch(returncode) {
  case 0: puts("try-code");
    longjmp(buf, 1); /* threw exc. 1*/ break;
  case 1: puts("exception handler 1"); break;
  default: puts("default exception handler"); break;
}
return 0;

Listing 2.4: Basic exception handling with setjmp.h

For added functionality, a simple data structure with exception info, such as an error message and non-int exception types can be also created. To provide support for nested exception handlers an exception stack, consisting of a set of frames, is needed [33]. Each frame in the exception stack corresponds to one try-catch statement (Figure 2.5), and contains a jmp_buf and possibly information to be used for control flow. When an exception is thrown, the necessary exception handler is found by unwinding the exception stack, and if none exists, the exception is uncaught and the program exits.

![Figure 2.5: An exception stack to track try-catch statements.](image)

### 2.2.2 The Exceptions4C Library

Several unknowns made it hard to estimate how much would be needed to provide an exception handling basis. Beyond applying the techniques described in the previous section, the basis needed to work with ENCORE’s concurrency model. To save time and avoid reinventing the wheel, an open-source library for exception handling, Exceptions4C, is used in this work.

Exceptions4C (E4C) is a small, portable, open source exception handling library that brings exception handling semantics C programs. The library is written by Guillermo Calvo¹, had its first release in 2009, and uses the techniques in Section 2.2.1 along with more advanced techniques to provide extended

¹The Exceptions4C library can be found at http://guillermocalvo.github.io/exceptions4c
functionality. There are two versions of the library, a minimal and straightforward light version, E4C light, and a more complex regular version, E4C regular, with extended exception handling functionality such as PTHREADS support. This PTHREADS support is the only feature of E4C regular that is of interest with regards to this thesis, and is explored when implementing exception handling in multithreaded settings (Section 4.5).

E4C features the language constructs try, catch, and finally, for finalization code, which are made available through macros. Macros for declaring exception types, and throwing and retrieving exceptions are also provided. E4C supports nested exception handlers, which are implemented through stack frames with dynamic scope. Thrown exceptions are represented as a C struct with an exception type, a message, a line number, and a file name. The exception types allow the programmer to use multiple exception handlers (catch clauses following a try block), instead of just a single universal handler, whereas the remaining data is primarily for debugging or logging purposes.

```c
#include <stdio.h>
#include <e4c_lite.h>

E4C_EXCEPTION(Exception1, "A message", RuntimeException);

int main() {
    E4C_TRY {
        print("Running try-code\n");
        E4C_THROW(Exception1, "Woops!");
    } E4C_CATCH(Exception1) {
        printf("Exception handler 1\n");
    } E4C_CATCH(RuntimeException) {
        printf("Default exception handler\n");
    } E4C_FINALLY {
        /* Finalize */
    }
    return 0;
}
```

Listing 2.5: Basic exception handling with Exceptions4C

**Exception Contexts** E4C stores important exception handling data in exception contexts. The exception contexts contain an array of frames, an array for jmp_bufs, the last thrown exception, a variable for tracking the number of used frames, and more.

```c
struct e4c_context {
    jmp_buf jump[E4C_MAX_FRAMES];
    struct e4c_exception err;
    struct{unsigned char stage; unsigned char uncaught;} frame[E4C_MAX_FRAMES + 1];
    int frames;
};
```

Listing 2.6: Exception contexts in E4C light.

Frames and jump buffers more or less have a one-to-one relationship, where the former simply provides additional information for control flow. Together, they, along with a frame number provide enough information for jumping to
appropriate code points during exception handling (Figure 2.6). E4C light and
E4C regular, without PTHREADS enabled, use one global exception context,
whereas E4C regular with PTHREADS enabled uses thread local contexts.

Figure 2.6: Using frames during exception handling

2.3 Related Work

There are numerous works on exception handling in languages with asynchronous
method calls. A subset of these are discussed in this section, with focus on works
in which futures are part of the exception handling model.

Dart  DART is a programming language developed by Google, used to build
web, server and mobile applications [8, 23]. The language is object-oriented,
class-based, and supports actor-style concurrency through isolates, which com-
unicate through message passing via futures or streams. Dart streams pro-
duce a sequence of values and through the yield statement adds a value to the
sequence [13].

DART provides try statements for exception handling using the keywords
try, on, catch, and finally. DART’s exceptions are objects, thrown using
the throw keyword, and the user may create new exception classes. The on
keyword prefixes exception handling code and allows catching exceptions of a
specific class and may be combined with the catch keyword, which catches
an exception of unspecified class and gives access to the underlying exception
object [13].

```dart
1 try { throw new Exception1(...); }  
2 on Exception1 { ... }  
3 on Exception2 catch(e) { print(e); }  
4 catch(e) { print(e); }  
5 finally { ... }

Listing 2.7: Exception handling in Dart
```

Exceptions when using futures can be handled using try statements, or by
registering a callback function catchError to method calls [23].
class Exception1 extends RuntimeException {}  // Exception1 is checked

class X {
  ...
  ReturnType method() {
    try {
      throw new Exception1();
    } catch (Exception1 e) { ... }
    catch (RuntimeException e) {
      ...
    } finally { ... }
  }
}

(b) Checked exceptions

void foo() throws Exception1 {
  ...
}

void bar() {
  try {
    foo();
  } // This handler must exist
  // for the program to compile
  catch (Exception1) { ... }
}

(a) try-catch-finally statements, custom exception classes

Listing 2.7: Exception handling in Java

Future<String> f = foo();
f.then((String s) => print(s)) .catchError((Error e) => handle(e));

Listing 2.8: Exception handling with futures in Dart

Java  Java is one of the most popular object-oriented programming languages of today, and has numerous libraries and frameworks to provide concurrency. There are two types of exceptions in Java, checked exceptions and unchecked exceptions [32]. The much debated checked exceptions must be explicitly handled in the program (Listing 2.7b), something that the compiler guarantees through a compile time check. Unchecked exceptions are not checked at runtime, may be left uncaught, and have RuntimeException as root class. Users can create custom exception classes by extending existing ones, and handle exceptions with try-catch-finally statements (Listing 2.7a).

The library java.util.concurrent provides many classes for concurrency, among others an implementation of futures [29, 36]. The get method of futures is a synchronization mechanism that blocks the running task and retrieves the computed value once available. Futures also have a cancellation method, which attempts to cancel a task. The get method can throw CancellationException, ExecutionException and InterruptedException, which can be handled by wrapping the method in a try statement (Listing 2.9).

Future f = ...  
try {
  f.get();
} catch (ExecutionException e) { ... }

Future f2 = ...
f2.cancel();
try {
  f2.get();
}

Java
The Google Guava project contains concurrency libraries and extends Java JDK’s Future class with `ListenableFuture` [6]. `ListenableFuture` allows the programmer to register callbacks to be run when the task is completed, which provide one method that is invoked upon a failed computation, and another that is invoked upon a successful computation.

**Scala** Scala is a Java-inspired programming language that features both object-oriented and functional programming [28, 30]. Exception handling is similar to that of Java, in that `try-catch-finally` blocks are used, exceptions are objects, and that custom exception classes can be created. Scala’s exception handling statements are however expressions, and its catch blocks define exception handlers through pattern matching anonymous functions.

Scala’s Try type (`scala.util.Try`) is used to represents a computation that may either be successful, or result in an exception [34]. This type is useful in functional programming, and is similar to the Either type of Scala as well programming languages such as HASKELL [31]. The result of a Try can be accessed through pattern matching, or through status and getter methods (Listing 2.10).

```scala
1 val result = Try ( throw new Exception1 )
2 result match {
3   case Success(s) => { println(s); }
4   case Failure(e) => {
5     e match {
6       case e: Exception1 => ...
7       case e: Exception2 => ...
8     }
9   }
10 }
11 // Without pattern matching
12 if (result.isSuccess) { println(result.get) }
13 else { println(result.failed.get) }
```

Listing 2.10: Exception handling in Scala

The Scala standard library provides an abstract implementation of future objects through `scala.concurrent.Future`. There are multiple ways to handle exceptions in futures. One way is to use the callback function `recover`, which is applied when an exception occurs in the callee. The programmer can define `recover`, and can through pattern matching specify exception handling behaviours for different exception classes (Listing 2.8a). Similarly the callback `onComplete` can be defined, which is applied when the future is completed and takes a Try object, instead of an exception, as argument (Listing 2.8b).

An alternative approach is to use the blocking synchronization mechanism `Await.result`, which waits until the future is ready and rethrows exception objects when the computation failed (Listing 2.8c).
Listing 2.8: Exception handling with futures in Scala

Actor-based concurrency in Scala is provided through the Akka framework (akka.actors), which is presented next.

Akka The Akka framework provides actor-based concurrency for both Java and Scala, and is used for building concurrent and distributed applications [18].

Similar to actors in Encore, the framework’s actors are objects with their own state and behaviour, and communicate through message passing. Actors communicate exceptions through futures, allowing the usual idioms of Java or Scala to be used during exception handling.

Another layer in Akka’s fault handling model is actor supervision. Actors are linked to another actor, a supervisor, with a fault handling strategy [2]. When a child actor throws an exception, the supervisor applies one of four supervision directives to the child:

- **resume** keeps the child’s internal state and resumes the child’s execution.
- **restart** clears the child’s internal states and restarts the child.
- **stop** permanently stops or kills the child.
- **escalate** fails the supervisor itself, and escalates the exception to a higher level supervisor.

There is a mapping from exception class to supervision directive, meaning the choice of directive depends on the exception class. Furthermore, the supervisor either applies the directive to just the one child (the OneForOneStrategy), or to the child as well as its siblings (the AllForOneStrategy). Note that actors form a hierarchy in Akka, something which Encore’s actors do not.

Erlang ERLANG is a concurrent, functional programming language designed from the ground up to create distributed, fault-tolerant, real-time systems that in principle never stop [3]. The language’s concurrency model is based on the
actor model, with message passing between lightweight processes, the ERLANG
equivalent to actors.

Fault-tolerance and error recovery is a high priority in ERLANG, which has
robust error handling mechanisms for both sequential and concurrent programs,
the latter of which will only be briefly touched upon. Sequential errors occur in
three classes:

- Runtime errors are produced when the program crashes, and have a rea-
  son such as badarg, badarith or undef attached. Though they are auto-
  matically raised, runtime errors can be emulated with error(Reason) or
  error(Reason, Args).

- Exit errors are generated errors, and occur when the code has a call to
  exit(Term).

- Throw errors are also generated, and indicate errors that the caller may
  want to catch. The code for throwing is throw(Term).

Listing 2.11: Error classes in Erlang

Exception handling happens in try expressions or catch expressions [15].
The former is built up of try, catch and after blocks, where after blocks en-
close finalization code. Through pattern matching and guards, errors of specific
classes and terms can be handled with try expressions (Listing 2.12).

Listing 2.12: try expressions in Erlang

The catch expression, not to be confused with catch blocks of try expres-
sions, returns the value of run code is returned unless an exception occurs.
Exceptions are caught, and returned along with additional information depending
on the error class.

Listing 2.13: catch expressions in Erlang
(a) Links. As process 2 dies, so do the linked processes 1 and 7, as well as 6.

(b) Supervision trees, hierarchies of supervisors and workers.

Listing 2.9: Connected processes in Erlang

Error Handling in Concurrent Programs  The philosophy behind Erlang’s error handling in concurrent programs is to let processes crash, and let other processes fix the problem [3]. Letting processes crash leads to less time spent writing defensive code, less resources spent trying to recover from errors, and serves to decouple error handling from application logic. Process have unique IDs, which can be used to connect them in different ways, so they can observe each other and take appropriate action when connected processes die.

There are three built-in ways to connect processes in Erlang, links, monitors and supervision trees. When a set of processes are linked and one process die, all linked processes are informed and also die (with some exceptions), throughout the entire connected graph (Figure 2.9a).

Monitors allow processes to be connected in a unidirectional way. When a monitored process dies, its parent, the monitor, is informed of this, but does not die. Monitors can thus handle errors when their child processes, as opposed to links that die in groups. Supervision trees allow hierarchical application structures to be created. A supervision tree consists of supervisors processes with child processes below them, which they supervise (Figure 2.9b). Like in Akka, supervisors have different strategies for handling errors in children, a restart strategy. One-for-one supervision trees restart just the single child when one dies, whereas one-for-all supervision trees kill and then restart all siblings when any child dies. Supervisors also carry specifications for each of their children, which includes information about how to start as well as shut down a child, when to restart a terminated child, and more.

Wrap-Up  All mentioned languages support exception raising and handling, either imperatively (e.g. try-catch statements) or functionally (e.g. Either types representing successful or failed computations). Erlang and Dart were built ground up with the actor-model, whereas Java and Scala utilize frameworks such as Akka for actor-based concurrency. Erlang’s takes a different approach in that it encourages the programmer to let processes crash and let
connecting processes, e.g. supervisors and monitors, fix the problem. Erlang does not have futures whereas the other languages do, though they are a recent addition in Java and Scala. The futures of these languages are similar to those of Encore, and provide exception handling through callback functions or through try-catch statements wrapped around retrieval of the future value.

The next chapter introduces the exception handling model that has been implemented for Encore.
3. Exception Handling in Encore

This chapter describes how exception handling works in Encore, using the implementation in this thesis. The exceptions handling model allows exceptions to be communicated in both synchronous and asynchronous settings. Section 3.1 introduces language constructs for raising exceptions of different types and exception handling expressions, which allow exceptions to be handled in synchronous settings. Asynchronous exception handling is exception handling that involves inter-process communication, communication of exceptions between actors, and is covered in Section 3.2.

3.1 Synchronous Exceptions

The exception handling mechanisms used in synchronous settings is inspired by Java, and features exception raising and handling. The exception handling mechanism uses type-based matching that also considers subtyping, and allows the programmer to create custom exception types. Statically checked exceptions, as found in Java [35], are not supported in this implementation, since compile time errors are outside of the scope of this thesis. Furthermore, an argument can be made that statically checked exceptions are not programmer-friendly due to the programming overhead.

Raising Exceptions: throw Exceptions are raised with throw statements, which are applied to an exception type and take an optional message.

```
1. throw MyException("Optional message")
2. throw MyException
```

Listing 3.1: Raising exceptions in Encore

Exception Handling Expressions: try Exceptions are handled through try-catch, try-finally or try-catch-finally expressions (Listing 3.1a). The try block encloses code that may raise an exception, and catch blocks define exception handlers for the possibly exceptional code. Multiple catch blocks may follow the try block, each one handling an exceptions of a different type, and optionally storing the exception to a variable when caught (Listing 3.1b).
The finally block represents finalization code, and must be the final clause in any exception handling expression. Finalization code executes at the end of the exception handling process, and does so whether or not an exception was raised.

Just like with ENCORE’s if-else expressions, the result of an exception handling expression can be stored to a variable (Listing 3.2). Note that the result of the expression will not be set to the result of the finally block, but to the result of the try block or a catch block.

**Custom Exception Types**: exception  The programmer can define custom exception types, which then become available to the exception handling mechanism (Listing 3.3). The new exception types are not available outside of exception handling, the way that int or string is usable. Each exception type is associated with a supertype and has a default message, and exceptions form a hierarchy, which gives the programmer freedom to handle exceptions through “umbrella types”, or specific types. The exception type RuntimeException is built-in, and is the root of the exception type hierarchy. Instances of a particular exception type can be thrown with a message, which, if omitted, is replaced by the default message of the exception type.

An important consideration is the scope of exception types. Class local exceptions were first considered, but as exceptions are often be communicated
between objects of different classes, they are not supported. Instead, ENCORE exception types simply have a global scope.

```haskell
1 exception MyException("Default message", RuntimeException)
2 fun foo() : unit
3 throw MyException("Woops!")
4 end
```

Listing 3.3: Defining exception types

Exception instances are objects of an exception class, with fields for exception data (Listing 3.4).

```haskell
1 try
2 throw MyException("Woops!")
3 catch e : RuntimeException
4 print(e.type)  -- "MyException"
5 print(e.message)  -- "Woops!"
6 end
```

Listing 3.4: Accessing exception data.

### 3.2 Asynchronous Exceptions

Actors need to be able to communicate exceptions to each other somehow. In the chosen exception handling model, exceptions are handled non-locally, and are propagated to the caller thread. A pull-model is used, where sender actors can handle exceptions when they see fit (Figure 3.2). This can be contrasted with using a push-model, where the receiver instead determines when exceptions should be handled. Any actor with a reference to an exception then needs to handle it, and not just the sender. This is relevant when an exception object is passed around by the sender.

![Push vs. Pull-model for exception handling.](image)

Figure 3.2: Push vs. Pull-model for exception handling.

Following the design suggested in *Fault in the Future* [22], futures are used to communicate exceptions between actors. Futures can thus hold exceptions
(a) Fulfilling futures with exceptions. Exception a is handled locally. Exception b has no local handler, and is propagated to a future.

(b) Handling exceptional futures

on top of normal values, and are fulfilled with an exception when the callee throws an exception without catching it (Figure 3.3a). This ends the execution of the callee’s currently running message, as if the message ran to completion, and allows the callee to execute other messages. If the executing message is a one-way message, where the caller is not requesting a return value, any uncaught exception are simply discarded.

Note that the main method, the entry point to every program, is not treated the same as other methods regarding exception handling. If an exception is thrown and left uncaught within the execution of main, it will remain uncaught and the program will crash. This is a conscious decision, and users must wrap possibly exceptional code in try expressions to protect against a crash.

As per the pull-model, the caller can handle the exceptional future at any time by using get. When retrieving a result from an exceptional future, get rethrows the exception. This means that any application of get on an exceptional future needs to be wrapped in an exception handling expression, (Figure 3.3b), or else the exception gets propagated again. If an exceptional futures or an exceptions is shared with other actors, they can in turn also handle the exception when they see fit. The mechanisms await and suspend remain unaffected from the programmers view, and can be used like normal.

This concludes the description of the exception handling model. In the next chapter, the implementation of the described model is explained.
4. Implementation

This chapter details the implementation details for the exception handling model (Chapter 3). Code generation is implemented for the new language constructs, and maps ENCORE code to C code that interacts with E4C in the runtime. An exception class is used to give users access to exception data, and to store in futures (Section 4.2). Actor communication through futures and changes to the get operation is explained in Section 4.3. To make sure that exceptions during do not enter the PONY runtime, and allow futures to be fulfilled with exceptions, the runtime and dispatch mechanism is modified (Section 4.4). Finally, ENCORE’s context swapping and E4C are modified in order to provide exception contexts in the appropriate scope (Section 4.5).

4.1 Language Constructs

This section outlines the implementation details of language construct specified in the exception handling model, with focus on code generation. Implementing these constructs will not involve any modifications to the ENCORE or PONY runtime, but requires changes throughout the compiler. in this section

E4C is well suited as a basis for synchronous exception handling, as it features C equivalents of the constructs specified in the exception handling model (Section 3). Once the language constructs are implemented, users will be able to use basic exception handling functionality in synchronous settings, and a base will be ready on which asynchronous exception handling will built. At this point E4C regular is given consideration, as it has built-in PThreads support.

Custom Exception Types: exception The first language construct to implement is the exception keyword, which is used to declare an exception type. The generated C code is an E4C macro, which in turn expands to the initial-
ization of a struct (Listing 4.1). In E4C, thrown exceptions include a reference to an exception type struct, and supertypes are references to such a struct.

To give exception types a global scope, the exception keyword is added as a top level construct, similar to the class keyword. This ensures that exception types can only be declared in the top level of the code. The generated C code
for all exception types is placed in a shared C file, which also ensures the same exception type is only declared once.

Raising Exceptions: `throw` Exception classes are thrown using the `throw` statement, which can take an optional message. The ENCORE code is translated to an invocation of the `E4C_THROW` macro.

Exception Handling Expressions: `try` The next construct is exception handling expressions (Listing 4.2), which are translated to C code in a straightforward manner thanks to the macros of the E4C library. Note that catch-all clauses are simply syntactic sugar for catch `RuntimeException`.

To allow the result of the exception handling expression to stored, a variable outside of the expression is used. At the end of the `try` block and `catch` blocks, the result variable is then set to the returned value (Listing 4.3). Note that this variable is not set in a `finally` block, adhering to the semantics of the language construct.
Listing 4.3: Code generation: Saving the result of exception handling expressions

```c
1 x = try
2 throw MyException("")
3 1
4 catch MyException
5 -1
6 finally
7 -2
8 end
```

4.2 Exception Objects

When exceptions are thrown, they are stored as C structs in the exception contexts of E4C. Each exception contexts stores only one exception, and they reside on the callee’s stack. Encore does not preserve the callee stack after a future is fulfilled, which means that accessing exceptions stored in futures is not memory safe (Figure 4.4).

![Figure 4.4: The callee stack is destroyed after a fulfill. References to the callee stack, are thus unsafe.](image)

To solve this problem and also allow the programmer to access the exception data in Encore, an exception class, ExceptionWrapper is created. It represents a thrown exception instance, and holds copies of the data in the E4C exception. This makes exceptions first class objects, and allow any amount of thrown exceptions to safely be stored in the user program.
ExceptionWrappers are created when a future is fulfilled (Section 4.4.2) or when an exception is caught, by simply desugaring catch clauses to fetch E4C data through embedded C code, and then passing it to the exception object constructor (Listing 4.3). The ExceptionWrappers are not shared between fulfilled futures and catch clauses, a limitation that is discussed in Section 6.1.1.

```plaintext
try
  ...
catch e : MyException
  -- Desugar catch clause by adding the following lines
  Type type = EMBED E4C_EXCEPTION.type END
  int line = EMBED E4C_EXCEPTION.line END
  -- Allocate string objects for 'char *' data
  String msg = new String(EMBED E4C_EXCEPTION.message END)
  String file = new String(EMBED E4C_EXCEPTION.file END)
  -- Pass fetched values to exception object constructor
  ExceptionWrapper e = new ExceptionWrapper(type, msg, line, file)
  -- User code
  ...
end
```

Listing 4.3: Creating exceptions objects in catch clauses.

Note that an ExceptionWrapper is allocated on the heap, and the same exception data exists in parallel on the stack until overwritten. A possible workaround to this duality is to modify E4C to use ExceptionWrappers and pass these by references, instead of using the single C struct. However, this adds an overhead to parts of the runtime, ENCORE’s dispatch mechanism (Section 4.4.2), which makes the chosen solution preferable.

Rethrowing Exceptions A rethrow method is created for the ExceptionWrapper class. This method simply calls E4C’s throw function, but uses the data of the ExceptionWrapper as arguments.

```plaintext
class ExceptionWrapper
  ...
  def rethrow() : unit
    -- Not using 'E4C_THROW' to pass file and line.
    EMBED(unit)
    e4c_throw(e4c_ctx(), #{this.e4c_type},
```

24
Listing 4.4: Rethrowing exceptions.

When the exception is later caught, new ExceptionWrappers are allocated with the same data as the rethrown exception. Just like with future fulfills and catch clauses, exception objects are not re-used during rethrows (also discussed in Section 5.3).

4.3 Communicating Exceptions Between Actors

As explained in Section 3.2, futures allow actors to share memory and communicate synchronously. To enable exceptions to be communicated between actors, futures are modified to be able to hold exceptions, on top of normal values. The future data structure is extended to store a pointer to exceptional data in the form of an ExceptionWrapper, and a flag that indicates whether they are fulfilled with an exceptional or normal value. The get is the only future operation which needs modification to accommodate for these changes. If a future is exceptional, get re-throws the exceptional value when operating on the future, and otherwise returns the value, as usual.

4.4 Protecting the Runtime and Future Fulfills

When an asynchronous method call results in an exception, E4C will kick in and carry out a non-local. However, no exception handler exists outside of the user code, which means the exception will propagate down to the PONY runtime and crash the program. A crash is only desired when the main method leaves an exception uncaught, for all other asynchronous method calls the exception must be caught in order to prevent a crash, and fulfill a possible future with the exception. The following sections give a background to ENCORE’s dispatch mechanism (Section 4.4.1) and presents a new version (Section 4.4.2), which protects the runtime and can fulfill futures with exceptions.
4.4.1 Encore’s Dispatch Mechanism

A dispatch mechanism looks up which version of a method or function to call. ENCORE uses *single dynamic dispatch*, where the properties of a single actor is used to find the proper function to run [26]. Under the hood, ENCORE’s dispatch mechanism also runs methods different ways depending a one-way or regular message is being processed (Section 2.1.1). To achieve this, each message in an actor’s inbox carries with it an ID that indicates the method to call and in what way. The dispatch mechanism then uses the message ID in a switch statement with one case per ID, and stores the result in the future of the message. This mechanism allows actors to postpone the execution of asynchronous method calls to any later point.

```cpp
void dispatch_MyClass(...) {
    switch (msg_id) {
    case REGULAR_MSG_method1: fulfill(future, method1(...)); break;
    case ONEWAY_MSG_method1: method1(...); break;
    ... }
}
```

Listing 4.5: Dispatch function of an ENCORE class. The dispatch function consists of a conditional, with multiple branches for each method, one corresponding to a different kind of call on the same method. Branches for regular messages involve fulfilling a future, whereas those of one-way messages are just a method call.

4.4.2 Modifying the Dispatch Mechanism

To deal with exceptions during asynchronous method calls, and store exceptional results in futures when processing regular messages, ENCORE’s dispatch mechanism needs to be modified. Currently, exceptional method calls, run from the dispatch function (Listing 4.5), do not have a fallback exception frame. When an exception is thrown and uncaught in the methods corresponding runtime function, the exception stack will be empty, leading E4C to exit the program.

![Diagram](image)

Figure 4.7: With ENCORE’s current dispatch mechanism, exceptional method calls result in the program exiting.

The dispatch mechanism has been modified to create a fallback exception frame, so that exceptions are caught when occurring in asynchronous method calls. This involves wrapping each call to the generated C functions of the methods in an exception handling statement, and adding additional in each switch case.
In switch cases of regular messages, the future is fulfilled with an `ExceptionWrapper` if an exception is caught, and the future’s `exceptional` flag (Section 3.2) is set to `true` by passing the value to a modified version of the fulfill function. In switch cases of one-way messages, caught exceptions are ignored since there is no return value, not letting the caller know what has happened. Alternative ways of dealing with exceptions in one-way message sends are discussed in Section 6.

```
void dispatch_MyClass(...) {
  switch (msg_id) {
    case REGULAR_MSG_method1:
      E4C_TRY { fulfill(future, method1(...), 0); }
      E4C_CATCH { fulfill(future, the_exception(), 1); }; break;
    case ONEWAY_MSG_method1:
      E4C_TRY { method1(...); }
      E4C_CATCH { /*do nothing*/ }; break;
    ... } }
```

Listing 4.6: Modifying dispatch to support exceptions. A boolean is passed to fulfill to indicate whether the value is exceptional. All method calls are enclosed by `try` expressions, in case an exception is left uncaught in the method.

### 4.5 Scope of Exception Contexts

Exception contexts carry crucial data which, if not preserved can lead to undefined behaviour in E4C. When context swapping occurs in Encore, another message or actor is takes over a thread, and if exception contexts are not swapped as Encore’s exception swapping mechanism is triggered, they will become corrupted. This section outlines modifications needed to Encore’s context swapping mechanism and E4C in order to provide exception contexts at the correct level.

E4C regular’s provides built in support for Pthreads, which enables each pthread to have its own exception context. However, exceptions on the level of pthreads will not work for Encore, where each pthread runs messages for many actors, and actors can migrate between pthreads as well as execute messages in an interleaving fashion (Section 2.1.3).

To properly preserve exception handling information during Encore’s context swaps, both actor blocking and interleaving of messages need to be considered. If only actor blocking occurred, then actors would process messages sequentially and only need an actor level exception context. However, with interleaved message execution, where message level information can be stored and resumed, each message needs its own exception context.

Note that message level exception contexts are needed in both multithreaded and single threaded settings. Consider the program in Listing 4.7, where a `suspend` is used during exception handling. If another actor then runs and creates an exception frame, the exception frame of the first actor is overwritten, leading to undefined behaviour.
class SeeFood
    def is_hotdog(s : String) : unit
        try
            suspend()
            if not s.equals("hotdog") then throw RuntimeException end
            println("Hotdog: ", s)
        catch
            println("Not hotdog: ", s)
        end
        end
end

class Main
    def main() : unit
        val fut_a = new SeeFood ! is_hotdog("avocado")
        val fut_h = new SeeFood ! is_hotdog("hotdog")
        get(fut_a)
        get(fut_h)
        end
end

Listing 4.7: Message level exception contexts in single threaded settings.

Solution  Since the PTHREADS support of E4C regular is not useful for ENCORE, E4C light is used in combination with ENCORE’s built in PTHREADS support. The ENCORE thread is extended with an e4c_context pointer, and a getter function is created to simplify accessing the new field. e4c_contexts are created and stored on messages’ stack as soon as they starts running, and the new field of the ENCORE thread is set to the address of the stored e4c_context.

To make sure E4C functions use the message level exception context instead of a global or thread local one, E4C functions and macros are modified to take a pointer argument to an e4c_context. By simply passing the message local e4c_context to the functions, E4C will use the correct exception context. The same exception context will naturally be used in the dispatch function (Section 4.4.2), and elsewhere in the runtime.

When the execution of a message resumes, the address of the message’s e4c_context, stored somewhere on its stack, has been lost, as the ENCORE thread now has a pointer to the e4c_context of the previous, possibly finished, message. To solve this and properly resume messages, a pointer to the e4c_context is created right before the context swap, and is then used to refresh the exception data when the a message resumes execution.
5. Evaluation

This section presents an evaluation of the exception handling implementation, and consists of an overview of its features (Section 5.1) and performance experiments (Section 5.2). Finally, some limitations of the implementation are discussed in Section 5.3.

5.1 Expressiveness

This section demonstrates features of the exception handling implementation, and how they simplify propagation and handling of errors in Encore. Note that basic testing has been successfully carried out on all demonstrated language constructs, to ensure that they behave as expected in both synchronous and asynchronous settings.

5.1.1 Synchronous exceptions

The exception handling mechanism simplifies error handling circumstances by allowing the programmer to use try-catch expressions and throw statements (Listing 5.1a). Without this implementation, the programmer needs to add extra logic to the program for propagating and handling exceptional values. As an example, objects of the Either type of Encore can be used to represent a computation that is either successful or a failure, and have to be handled in a match statement in order to mimic exception handling behaviour (Listing 5.1b).

Custom exception types can be defined by the user, and can then be used to distinguish between different kinds of exceptions. Furthermore, exceptions form a hierarchy, which allows the programmer to arbitrarily group exception types during exception handling (Listing 5.2b). To distinguish between different kinds of exceptions without the implemented exception handling mechanism, the programmer would have to map values to exceptions and use conditional statements (Listing 5.2b). Handling exception hierarchies properly would require even more control logic.

With the implemented exception handling mechanism, exceptions can also be saved to variables when caught, giving the user access to additional information (Listing 5.1). The saved exception is a first class object that can be passed around, and provides a rethrow method.
try
if all_is_fine then
  1
else
  throw RuntimeException
end
catch
  -- error
end

(a) The implemented try expressions

val result =
if all_is_fine then
  Right[int,int](1)
else
  Left[int,int](-1)
end

(b) Mimicking try functionally

Listing 5.1: Evaluation of try expressions

exception IOExc("", RuntimeException)
exception AccessDenied("", IOExc)
exception DirNotFound("", IOExc)
exception FileNotFoundException("", IOExc)

... try
if all_is_fine then
  1
else
  if accessdenied then
    throw AccessDenied
  else
    throw IOExc
  end
catch AccessDenied
  request_access()
end
catch IOExc
  ask_for_new_path()
end

match result with
  case Right(x) =>
    x
  end

... case Left(e) =>
  -- error
  end
  end

(a) With exception types
(b) With complex control logic

Listing 5.2: Distinguishing between different types of errors.

... try
if all_is_fine then
  1
else
  if accessdenied then
    throw AccessDenied
  else
    throw IOExc
  end
catch AccessDenied
  request_access()
end
catch IOExc
  ask_for_new_path()
end

match result with
  case Right(x) =>
    x
  end

... case Left(e) =>
  -- error
  end
  end

(a) With exception types
(b) With complex control logic

Listing 5.2: Distinguishing between different types of errors.
val fut = actor ! foo()
try
get(fut)
catch
  -- handle
end

(a) Through get and try-catch

val res = get(actor ! foo())
match res with
  case Right(x) =>
    ...
  case Left(x) =>
    -- handle
end

(b) Functionally through Either types

val fut = actor ! foo()
await (fut)
this.suspend()
try
  println(get(fut))
catch
  -- fut contained exception
end

(c) Cooperative scheduling primitives are used like usual.

Listing 5.3: Handling exceptions in asynchronous settings

fun exceptional(x : int) : unit
try
  if x > 0 then throw RuntimeException("neg") end
  print(x)
catch e : Exception1
  print("Message: ", e.message)
e.rethrow()
end
end

Listing 5.1: Accessing exception objects.

5.1.2 Asynchronous Exceptions

In this implementation, there is no significant difference between how exceptions are handled asynchronously and synchronously. Futures are used as a middleman, and the get operation, which now rethrows exceptions in futures, has to be wrapped in try-catch expressions in order to handle the exception. Without these changes, Either would have to be used, like in the synchronous case (Listing 5.3b). The cooperative scheduling primitives await, get and suspend all work as expected, without requiring any extra work on the programmer side (Listing 5.3c).
5.2 Performance

This section presents CPU running time experiments for the implemented exception handling mechanism. The first motivation behind the experiments was to get a rough idea about the runtime overhead caused by ENCORE’s passive use of exception handling (e.g. during context swapping and dispatch). The second motivation was to see how programs scale as more exception handling related operations are executed. Section 5.2.1 presents performance experiments mainly aimed at learning more about the overhead of the exception handling mechanism, and Section 5.2.2 at learning how well the mechanism scales. The results and concluding remarks about the experiments are presented in Section 5.2.3.

All tests were run under Ubuntu 14.04 64-bit through VirtualBox on a Windows 8.1 host. The host had a 2.00 GHz Intel Core i7 4510U with 512 kB L2 cache and 8 GB RAM, out of which 1.5 GB was made available to the guest environment. Running times of programs were measured with the Linux `time` command, and each test configuration was run at least three times. The time was calculated by averaging the `user` and `sys` measurements output by `time`. Profiling data of programs was also gathered, using the GNU profiler `gprof`, and was used to pinpoint resource hogs. Different test programs were run under slightly different circumstances, and the figures are not meant to be used in to compare the test programs, but to compare runs of the same test program with varying parameters.

5.2.1 Runtime Overhead Experiments

Even when user programs do not use exception handling, it is used by ENCORE when messages are run, during context swapping, and in dispatch. To get a rough idea about the runtime overhead several test programs are executed on a version of ENCORE where exception handling is disabled, and on a version where exception handling is enabled. The running time of the test runs are then measured, and compared.

**Basic Message Send Tests** The first set of programs do basic message sends from the main actor to one of four actors. Main calls the method `foo` of the actors, which simply returns 1. The call to `foo` is done in three ways, in Test1 it is a simple one-way message send, in Test2 the result is saved to a future, and in Test3 the result is retrieved with `get` (Listing 5.2). Every program is run with varying values for the `times` parameter.

```plaintext
active class A
  def foo() : int
    1
  end
end

active class Main
  def main(args:[String]) : unit
  var times = get_times(args)
```
var nactors = 4
var a = new [A](nactors)
repeat i <- nactors do
  a(i) = new A()
end

repeat i <- times do
  -- Test 1: One-way message sends
  a(i%nactors) ! foo()
  -- Test 2: Regular messages
  var f = a(i % nactors) ! foo()
  -- Test 3: Synchronizing with 'get'
  get(a(i%nactors) ! foo())
end
d
Listing 5.2: Three basic performance tests that do not use exception handing.

Savina Tests The second set of programs are existing stress tests from the Encore test suite (found as appendices in Chapter A), and involve more complex interactions than the basic message send tests. Some of the Savina tests mainly do one-way message sends in the end, whereas others involve a lot of regular message sends and object allocation. Only one of configuration for each Savina programs is run.

5.2.2 Scalability Experiments
The performance experiments in this section are meant to show how well a program performs as more exception handling operations are executed. The programs do basic message sends from the main actor to one of four actors. Main wraps the method call in a try-catch expression and either calls foo, which simply returns 1, or throw, which throws an exception. One of five programs is run, each of which use exception handling mechanisms in a slightly different way (Listing 5.4). Each program is run with varying values for the times parameter.

5.2.3 Results
Given the results from Test 1, as well as some of the Savina tests, there seems to be a significant overhead when exception handling is not utilized. On Test 1, exception handling causes a 13% slow down with 40 M message sends, and a 28% slow down with 100 M message sends, and the running time seems to increases exponentially (Table 5.1). Similarly, the Savina tests Pingpong, Counting, Chamaneos and Big also indicate that there is significant overhead with exception handling enabled (Table 5.3).

On the other hand, experiments on Tests 2 and 3 did not result in as severe overheads, but these programs could only be executed with a maximum of 1 M message sends (Figure 5.2). The reason for this limitation is that these programs
Listing 5.4: Basic test programs where exception handling is used within the user program. Tests 4 through 7 catch without saving. Test 8 is like Test 7, but saves caught exceptions.
<table>
<thead>
<tr>
<th>Messages (millions)</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1: No exceptions [s]</td>
<td>1.4</td>
<td>1.9</td>
<td>2.2</td>
<td>4.5</td>
<td>8.8</td>
<td>13.2</td>
<td>18.0</td>
<td>22.8</td>
</tr>
<tr>
<td>Test 1: Exceptions [s]</td>
<td>1.4</td>
<td>1.9</td>
<td>2.3</td>
<td>4.6</td>
<td>9.9</td>
<td>15.5</td>
<td>21.8</td>
<td>29.3</td>
</tr>
<tr>
<td>Overhead</td>
<td>1%</td>
<td>4%</td>
<td>5%</td>
<td>3%</td>
<td>13%</td>
<td>18%</td>
<td>21%</td>
<td>28%</td>
</tr>
</tbody>
</table>

Table 5.1: Test1 running time (in seconds, rounded up) with and without exceptions enabled, for varying amounts of message sends, and the calculated overhead. The test sends one-way messages, and does thus not involve allocation of any futures. A noticeable overhead is observed when exception handling is enabled, which seems to increase exponentially.

<table>
<thead>
<tr>
<th>Messages (thousands)</th>
<th>400</th>
<th>600</th>
<th>800</th>
<th>900</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 2: No exceptions [s]</td>
<td>0.6</td>
<td>0.8</td>
<td>1.1</td>
<td>1.2</td>
<td>1.5</td>
</tr>
<tr>
<td>Test 2: Exceptions [s]</td>
<td>0.6</td>
<td>0.9</td>
<td>1.1</td>
<td>1.3</td>
<td>1.5</td>
</tr>
<tr>
<td>Overhead</td>
<td>5%</td>
<td>16%</td>
<td>5%</td>
<td>6%</td>
<td>2%</td>
</tr>
<tr>
<td>Test 3: No exceptions [s]</td>
<td>1.4</td>
<td>2.1</td>
<td>2.8</td>
<td>3.1</td>
<td>3.5</td>
</tr>
<tr>
<td>Test 3: Exceptions [s]</td>
<td>1.5</td>
<td>2.2</td>
<td>2.9</td>
<td>3.4</td>
<td>3.8</td>
</tr>
<tr>
<td>Overhead</td>
<td>5%</td>
<td>7%</td>
<td>5%</td>
<td>9%</td>
<td>7%</td>
</tr>
</tbody>
</table>

Table 5.2: Test2 and Test3 running time (in seconds, rounded up) with and without exceptions enabled, for varying amounts of message sends, and the calculated overhead. Both tests involve sending regular messages, where futures are created. Test2 involves messages such as `var f = a!foo()`, and Test3 `get(var f = a!foo())`

allocate a future for every message, and quickly cause garbage collection to trigger. Tests 5, 7 and 8 are similar, in that they allocate a future for every message, and also hit a roof at around 1 M messages. Tests 4 and 6 are similar to Test 1, in that they send one-way messages (Figure 5.5a). One thing to notice is that Test 6, which also throws exceptions, has a significantly higher running time than Test 4.

The profiling data verifies that large amounts of regular messages trigger garbage collection, and also shows that the stack unwinding of the exception handler (try-catch in ENCORE’s dispatch) causes much slow down. Note that these tests are worst-case in the sense of only executing trivial behaviours (e.g. directly returning an integer). This means exception handling, context switching, queuing messages, handling messages, carrying out function calls, and so on, have a huge overhead from a percentage point of view. As an example, the gprof data shows that Test 4 spends around 30% of the time queuing and handling messages, 5% dispatching functions, and over 9% on exception handling. In real applications, methods are commonly more long-running, meaning the fraction of time spent managing exceptions will be reduced.
Table 5.3: Savina tests running time (in seconds, rounded up) with and without exceptions enabled, for a fixed number of operations, and the calculated overhead.

<table>
<thead>
<tr>
<th>Test Argument</th>
<th>Pingpong 300 K</th>
<th>Counting 50 M</th>
<th>Fib 26</th>
<th>Chameneos 800 K</th>
<th>Big</th>
<th>BndBuffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions [s]</td>
<td>6.7</td>
<td>10.0</td>
<td>2.9</td>
<td>14.7</td>
<td>22.0</td>
<td>16.9</td>
</tr>
<tr>
<td>No exceptions [s]</td>
<td>7.3</td>
<td>15.4</td>
<td>2.7</td>
<td>16.9</td>
<td>30.8</td>
<td>16.9</td>
</tr>
<tr>
<td>Overhead</td>
<td>10%</td>
<td>54%</td>
<td>-8%</td>
<td>15%</td>
<td>40%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Figure 5.5: Results of Tests 4 through 8, tests program with involving exception handling.

As a last minute alternative, a modification was made to avoid stack-unwinding when exceptions were not in use. Instead of non-local jumps when throwing exceptions to dispatch, a flag within the exception context is set to indicate that the exception was not propagated, and the method call returns as usual. Dispatch then knows whether an exception was raised by checking the propagation flag, and if so fulfills with the exception. The profiling data of the modified version looks promising, and spends significantly less time with exception handling functions, though thorough analysis has not yet been carried out.

5.3 Limitations

This section discusses limitation of the implementation and doubles as a future work within the scope of the chosen exception handling model. Future work outside of the exception handling model, as well as suggestions for optimization are brought up in Section 6.1.

Some of the trivial compiler level issues, relating to syntax and code generation, are brought up in Section 5.3.1. Streaming methods and future chaining, two ENCORE mechanisms which the exception handling library does not yet support, are discussed in Section 5.3.2. The last limitation in the implementation relates to a limit on the number of exception frames, and is discussed in Sections 5.3.3.
5.3.1 Compiler Level Issues and Refactoring

Below is a list of issues related to code generation, syntax and more, most of which are trivial to solve. Though the issues may be trivial, they are useful for the ENCORE team to know about.

- Exception types are not known to ENCORE, but only exist in the underlying C code. To fix this, ENCORE’s program table needs to be modified to hold information about exception types. Each use of the exception construct will then add the new exception type to the program table. Exception objects currently hold references to an E4C exception type represented by a C struct. This reference can be replaced with a reference to the exception type in the modified program table.

- The throw statement can only be passed string literals throw MyException ("Goodbye world!") and not string objects or expressions throw MyException (msgString). Fixing this requires modifying throw to take an expression, instead of string literal, as second argument, and during typechecking make sure that the expression evaluates to a string.

- The function name of the ExceptionWrapper constructor is hard coded in constructor calls, which is not very robust. If possible, the function name should be generated by the compiler.

- Exception type definitions can be more expressive by making arguments optional. The supertype could default to RuntimeException and the default message to the name of the exception type.

```java
exception MyExc1
would become desugared to
exception MyExc1("MyExc1", RuntimeException).
```

5.3.2 Streaming Methods and Future Chaining

Due to time constraints, streaming methods and future chaining have been ignored in this thesis, which need to be dealt with at some point. In short, streaming methods return streams, which represents a sequence of values that will be produced. The yield statement is used to generate one value at a time in the stream’s sequence. Streams are used in a similar manner to futures, but also have the eos operation which checks if the stream has reached its end, and the getNext operation, which returns the next value in the stream’s sequence.

```java
active class MyStreamer
stream even(): int
var i = 0
while i < 10 do
  yield(i) -- add i to sequence
  i = i + 2 -- prepare next value
end
end
```
active class Main
def main() : unit
  var even = new MyStreamer! even()
  while not (eos (even)) do
    println (get (even))
    even = getNext (even)
  end
end
end

Listing 5.3: A basic use case of streams.

One possible solution to get exception handling to work with streaming methods is to modify streams to how futures have been modified. The stream is then extended to carry exceptions and an indicator flag, and the get (and perhaps getNext) operation of streams is modified to rethrow exceptional values. The put operation, which is the stream equivalent of fulfill, is modified to take an extra argument that indicate that a produced value is exceptional. Finally, dispatch clauses of stream methods, similarly to those in regular or future asynchronous method calls, is wrapped in an exception handling statement.

A possible solution for allowing exception handling to work with future chaining is to modify the chain operation in a similar manner to dispatch clauses of regular messages. Function calls are then wrapped in an exception handling statement, and if caught, the future is fulfilled with an exception. This is possibly the only change that is needed.

5.3.3 Exception Frame Limit

E4C’s exception contexts store exception frames in an array, which means that the depth of nested exception handling statements in executing messages must not exceed some fixed value, currently five. This is not problematic from a practical standpoint as it is unusual and unnecessary that a program, in the context of one message, recursively handles exceptions to a significant depth. If exceptions need to be handled in a deep recursion, it is enough to use a single exception handler accompanied with control logic.

Having a fixed number of exception frames leads to an memory overhead when all exception frames are not used. The size of all frames is stack_limit * (jmp_buf_size + 2) bytes. Assuming a jmp_buf_size = 200 bytes, there is a 202 byte overhead for each unused frame. The depth of five is arbitrary, and could perhaps be reduced by a level or two, though it is uncertain whether this is a meaningful optimization. Perhaps it is possible to use a dynamic data structure instead, circumventing both the depth and memory overhead issue.
6. Conclusion

**Encore** now has a basic exception handling mechanism that is suitable for its concurrency model. The exception handling mechanisms used in synchronous settings is inspired by **Java**, and features exception raising and handling, through **try-catch-finally** expressions. It uses type-based matching that also considers subtyping, and allows the programmer to define and throw custom exception types. Thrown exceptions can be accessed in the form of objects of class **ExceptionWrapper**, which are assigned exception types, carry messages, can be shared between actors, and can be rethrown.

A pull-model of exception handling is used for inter-process communication of exceptions, allowing exceptions to be propagated from one actor to another through futures, and only need handling when the **get** mechanism is applied on a future.

A modified version of the open-source C library **Exceptions4C** serves as basis for the exception handling implementation, and provides non-local jumping functionality. **Exceptions4C** carries exception data in the form of exception contexts, which if not preserved leads to undefined program behaviour. Due to **Encore**’s scheduling constructs **await** and **suspend**, each running message must store its own exception context, which adds to the complexity of the exception handling mechanism. Each message carries an exception context in its stack, and **Encore**’s context swapping mechanism swaps message contexts whenever a message continues execution, or when the aforementioned scheduling constructs are applied.

The exception handling implementation works as intended, but carries a certain overhead, even when not directly used in a program. The main cause behind this is possibly **try-catch** functionality used in **Encore**’s dispatch mechanism, which occurs whether or not the program uses exceptions. In worst-case test programs, in the sense of only executing trivial, short-running methods (e.g. directly returning an integer), the overhead of the exception handling implementation was often over 10%. In other test programs, where a lot of futures were used to propagate results, the overhead was far less noticeable. It is hard to predict an overhead for real applications, but it will be low in programs with long-running methods or that are memory intensive, and high when there is lots of inter-thread communication and short-running methods.

In hindsight, dealing with exception contexts may seems easy, but reaching this point required learning a lot about **Encore**'s concurrency model and
scheduler. Some important questions in reaching the solution are:

- How do threads work in the language?
- Is there a one to one mapping between actors and threads?
- Do running messages keep a stack, or do they share an actor level stack?
- What constructs can control scheduling in ENCORE, and what are their semantics? (This includes synchronization mechanisms.)
- When does context swapping occur in the language, and how does this related to the scheduling constructs?

In the case of ENCORE, actors do not map to threads, and because of the non-blocking synchronization await or the suspend operation, context swapping can occur in the middle of message execution. Each message must thus have its own stack, and exception contexts must be saved individually for each message.

6.1 Future Work

Section 6.1.1 outlines an optimization for allocation of ExceptionWrappers, which is currently done in a wasteful manner, and Section 6.1.2, two approaches to letting users attach data to exceptions.

6.1.1 Redundant Allocation of Exception Objects

To avoid needlessly allocating objects, exceptions exist as C structs in the runtime when thrown, and occur as ENCORE objects (ExceptionWrappers) in the user program only to provide user accesses and fulfill futures with an exceptional value. However, this design choice along with time constraints of this thesis has led to some limitations in exception objects, one of which can be remedied while keeping this duality, and another which requires a fundamental redesign to be fixed.

ExceptionWrappers, are not re-used or shared between futures, catch clauses and rethrows (Section 4.2), but redundant copies are created at each point an exception is moved from a future to a catch clause, or from a rethrow to a future. To re-use exception objects and optimize the exception handling runtime, an ExceptionWrapper pointer can be added to the E4C context, which the rethrow method, dispatch clauses and catch clauses interact with. The rethrow method sets the pointer to the address of the target ExceptionWrapper, before throwing an exception as usual.

```c
class ExceptionWrapper
...
def rethrow() : unit
  EMBED(unit)
  -- set ExceptionWrapper pointer to address of this object.
  e4c_set_exceptionwrapepr(#(this));
```
Listing 6.1: Modifying the rethrow method to share the current \texttt{ExceptionWrapper}

Dispatch and \texttt{catch} clauses can then be modified to re-use the pointer, unless \texttt{NULL}, instead of allocating new \texttt{ExceptionWrappers}.

Listing 6.2: Modifying the dispatch to re-use existing \texttt{ExceptionWrappers}.

Listing 6.3: Modifying \texttt{catch} clauses to re-use existing \texttt{ExceptionWrappers}.

6.1.2 Custom Data in Exceptions

Users may want to attach custom data to exceptions, which is not possible in the current implementation. Two approaches to this problem are discussed, both with the prerequisite that the runtime can re-use exception objects as explained in Section 6.1.1.

\textbf{Approach 1: Adding a Data Member to Exception Objects} Taking inspiration from .NET exception’s \texttt{Data} property [12], a dynamic data structure along with getter and setter methods is added to the \texttt{ExceptionWrapper} class. To attach custom data to an exception, users create the exception object before throwing it through the \texttt{ExceptionWrapper} constructor. Data is added with the setter method, and the \texttt{rethrow} method is then used to throw it.
Approach 2: User Defined Exception Classes  Another approach is to allow user to throw instances of user defined exception classes, as in JAVA [29] and C#, instead of the current exception types. In ENCORE this can be achieved through traits, which provides a set of methods, and also describe a set of required methods and fields to be included in any class that wish to include the trait [9]. By creating a trait throwable, the shared fields and methods of exceptions can be required, and methods such as rethrow can be provided, allowing the runtime to interact with the custom exception class in a predictable way. Users can then add any custom fields and methods to their exception class, which can be used for any purposes during exception handling. Note that the supertype matching may turn out to be problematic with this approach.

6.1.3 The Exception Handling Model

In languages such as DART, JAVA and SCALA, callbacks for exception handling can be registered to futures or method calls. One problem with ENCORE’s one-way messages is that there is no way for the caller to know whether or not the called method was successfully executed. Allowing callbacks to be defined and run when methods fail or succeed, could elegantly solve this problem.

Exceptions for Parallel Types  ENCORE offers support for parallel types, which are collections of data operated on in parallel [16]. The operations applied to elements of collections are functional, and among others include mapping a
function to each element, filtering, and returning the first available result followed by killing all other parallel computations. Exception handling for parallel types can be implemented by providing new operations that work on an `Either` data type provided in this thesis, which is a data structure containing a normal value or an exception. The new operations can allow failing the parallel if an element fails, replacing failed elements with a default values, filtering failed elements, and more.
Bibliography


Appendices
A. Savina Tests

A.1 Pingpong

Listing A.1: 1.pingpong/Main.enc

```
active class PingActor
  var pingsLeft : int
  def init(count : int, pong : PongActor) : unit
    this.pingsLeft = count
    pong!sendPing(this)
    this.pingsLeft = this.pingsLeft - 1
  end
  def sendPong(pong : PongActor) : unit
    if this.pingsLeft > 0 then
      pong!sendPing(this)
      this.pingsLeft = this.pingsLeft - 1
    end
  end
end

active class PongActor
  var pongCount : int
  def init() : unit
    this.pongCount = 0
  end
  def sendPing(ping : PingActor) : unit
    print("Got ping {}\n", this.pongCount)
    ping!sendPong(this)
    this.pongCount = this.pongCount + 1
  end
end

active class Main
  def main(args : [String]) : unit
    if |args| != 2 then
      print("input required: number of pings not specified\n")
    else
      let
        N = match (args(1)).to_int() with
          case Just(result) => result
```
A.2 Counting

Listing A.2: 3.Counting/Count.enc

```plaintext
active class Counter
  var count : int
  def init() : unit
    this.count = 0
  end
  def increment() : unit
    this.count = this.count + 1
  end
  def retrieve(p : Producer) : unit
    p!resultMessage(this.count)
  end
end

active class Producer
  var counter : Counter
  var iterations : int
  def init(counter : Counter, iterations : int) : unit
    this.counter = counter
    this.iterations = iterations
  end
  def increment(max : int) : unit
    var i = 0
    while i < max do
      this.counter!increment()
      i = i + 1
    end
    this.counter!retrieve(this)
  end
  def resultMessage(count : int) : unit
    if this.iterations != count then
      print("ERROR: expected: {}, found: {}, this.iterations, count")
    else
      print("SUCCESS! received: {}, count")
    end
  end
end

active class Main
```
def main(args : [String]) : unit
    if |args| != 2 then
        print("number of pings not specified\n")
    else
        let
            n = match (args(1)).to_int() with
            case Just(result) =>
                result
            end
            case Nothing =>
                print("number of pings not specified\n")
                0
            end
        end
        counter = new Counter()
        producer = new Producer(counter, n)
        in
            producer!increment(n)
        end
    end
end

A.3 Fib

Listing A.3: 6.Fib/fib.enc

import Std

linear trait Action
    require var msg : String
    require var number : int
def getMsg() : String
    this.msg
end
def setNumber(n : int) : unit
    this.number = n
end
def getNumber() : int
    this.number
end
end

active class Main
    def main(args : [String]) : unit
        if |args| != 2 then
            print("input required: number-th of fibo not specified\n")
        else
            let
                N = match (args(1)).to_int() with
                case Just(result) =>
                    result
                end
                case Nothing =>
                    print("number of pings not specified\n")
                    0
                end
            end
            counter = new Counter()
            producer = new Producer(counter, N)
            in
                producer!increment(N)
            end
        end
    end
end
let
  fjRunner = new FibonacciActor(Nothing)
in
  fjRunner!process(new Request(N))
end
end
end
end
end

linear class Request : Action
var msg : String
var number : int
def init(n : int) : unit
  this.msg = "Request"
  this.number = n
end
def request(n : int) : unit
  this.number = n
end
end

linear class Response : Action
var msg : String
var number : int
def init(value : int) : unit
  this.msg = "Response"
  this.number = value
end
def response_one() : unit
  this.number = 1
end
end

active class FibonacciActor
var result : int
var respReceived : int
var parent : Maybe[FibonacciActor]
def init(parent : Maybe[FibonacciActor]) : unit
  this.result = 0
  this.respReceived = 0
  this.parent = parent
end
def process(msg : Action) : unit
  match msg.getMsg() with
  case "Request" =>
    if msg.getNumber() <= 2 then
      this.result = 1
      this!processResult(new Response(1))
    else
      var request = new Request(msg.getNumber() - 1)
      val fi = new FibonacciActor(Just(this))
val f2 = new FibonacciActor(Just(this))

end

end

def processResult(var response : Response) : unit
match this.parent with
  case Just(p) => { p!process(consume response); (); }
  case Nothing => print(" Result = {}

end

end
end

A.4 Chameneos

Listing A.4: 7.Chameneos/Chameneos.enc

-- Colour convention:
-- Blue  = 1
-- Red   = 2
-- Yellow = 3

import List

fun doCompliment(c1 : int, c2 : int) : int
if c1 == 1 && c2 == 1 then
  1
else if c1 == 1 && c2 == 2 then
  3
else if c1 == 1 && c2 == 3 then
  2
else if c1 == 2 && c2 == 1 then
  3
else if c1 == 2 && c2 == 2 then
  2
else if c1 == 2 && c2 == 3 then
  1
else if c1 == 3 && c2 == 1 then
  2
else if c1 == 3 && c2 == 2 then
  1
else if c1 == 3 && c2 == 3 then
  52
3
else
  abort("This should never happen!")
end
end

fun lookup(n : int) : String
if n == 0 then
  "zero"
else if n == 1 then
  "one"
else if n == 2 then
  "two"
else if n == 3 then
  "three"
else if n == 4 then
  "four"
else if n == 5 then
  "five"
else if n == 6 then
  "six"
else if n == 7 then
  "seven"
else if n == 8 then
  "eight"
else if n == 9 then
  "nine"
else
  abort("this should not happen!")
end
end

fun spell(n : int) : List[String]
  var spelledList = new List[String]()
  if n == 0 then
    spelledList.prepend(lookup(n))
  spelledList
else
  var remaining = n
  while remaining > 0 do
    spelledList.prepend(lookup(remaining % 10))
    remaining = remaining / 10
  end
  spelledList
end
end

fun spellAndPrint(n : int) : unit
  var spelledList = spell(n)
  var cursor = spelledList.first
  while cursor != null do
    print("{0} ", cursor.getData().getValue())
    cursor = cursor.getNextLink()
  end
end

active class Creature
var place : MeetingPlace
var colour : int
var id : int
var sameCount : int
var count : int
var silent : bool

def init(place : MeetingPlace, colour : int, id : int, silent : bool) : unit
    this.place = place
    this.colour = colour
    this.id = id
    this.sameCount = 0
    this.count = 0
    this.silent = silent
end

def meet(id : int, colour : int) : unit
    this.count = this.count + 1
    this.colour = colour
    if this.id == id then
        this.sameCount = this.sameCount + 1
    end
    this.place!meet(this, this.id, this.colour)
end

def stop() : unit
    unless this.silent then
        print("{} {}
", this.count, this.sameCount)
    end
    this.place!sumMeetings(this.count)
end

def run() : unit
    this.place!meet(this, this.id, this.colour)
end

active class MeetingPlace
var meetingsLeft : int
var firstColour : int
var firstID : int
var firstChameneos : Maybe[Creature]
var meetings : int
var amountOfCreaturesDone : int
var numberOfCreatures : int
var main : Main
var firstRun : bool
var started : bool

def start() : unit
    this.started = true
end

def init(meetingsLeft : int, main : Main, numberOfCreatures : int, firstRun : bool) : unit
    this.meetingsLeft = meetingsLeft
```python
this.firstColour = -1
this.firstID = 0
this.firstChameneos = Nothing
this.amountOfCreaturesDone = 0
this.main = main
this.numberOfCreatures = numberOfCreatures
this.firstRun = firstRun
this.started = false
end

def sumMeetings(meetings : int) : unit
  this.meetings = this.meetings + meetings
  this.amountOfCreaturesDone = this.amountOfCreaturesDone + 1
  if this.amountOfCreaturesDone == this.numberOfCreatures then
    spellAndPrint(this.meetings)
    if this.firstRun then
      this.main!runSecondScenario()
    end
  end
end

def meet(chameneos : Creature, id : int, c : int) : unit
  if not(this.started) then
    this!meet(chameneos, id, c)
  else if this.meetingsLeft == 0 then
    chameneos!stop()
  else if this.firstColour == -1 then
    this.firstChameneos = Just(chameneos)
    this.firstColour = c
    this.firstID = id
  else
    val newColour = doCompliment(c, this.firstColour)
    this.firstColour = -(1)
    this.meetingsLeft = this.meetingsLeft - 1
    chameneos!meet(this.firstID, newColour)
    match this.firstChameneos with
      case Just(cham) => cham!meet(id, newColour)
      case Nothing => abort("ERR. There is not initial chameneos")
    end
  end
end

active class Main
  var meetings : int

def runSecondScenario() : unit
  val colours = [1, 2, 3, 2, 3, 1, 2, 3, 2, 1]
  this!runDefault(this.meetings, colours, false)
end

def runDefault(n : int, colours : [int], firstRun : bool) : unit
  val place = new MeetingPlace(n, this, [colours], firstRun)
  repeat i <- [colours] do
    print("{} ", colours(i))
  end
end
```
println"
repeat i <- |colours| do
  val colour = colours(i)
  val creature = new Creature(place, colour, i, false)
  creature!run()
end
place!start()
end

def run(meetings : int, creatureAmount : int) : unit
  val place = new MeetingPlace(meetings, this, creatureAmount, false)
repeat i <- creatureAmount do
  val colour = i % 3 + 1
  val creature = new Creature(place, colour, i, true)
  creature!run()
end
place!start()
end

def printColoursAux(c1 : int, c2 : int) : unit
  print("{} + {} -> {}
end

def printColours() : unit
  this.printColoursAux(1, 1)
  this.printColoursAux(1, 2)
  this.printColoursAux(1, 3)
  this.printColoursAux(2, 1)
  this.printColoursAux(2, 2)
  this.printColoursAux(2, 3)
  this.printColoursAux(3, 1)
  this.printColoursAux(3, 2)
  this.printColoursAux(3, 3)
  print("\n")
end

def extractor(maybe : Maybe[int]) : int
  match maybe with
  case Nothing => -1
  case Just(i) => i
end
end

def main(argv : [String]) : unit
  if |argv| == 1 then
    -- This was written because the results are not deterministic and the CI should at least check that it compiles
    println("Please supply arguments. Defaults should be '800000' for first argument")
    exit(0)
  end
  val colours = [1, 2, 3]
  val numberOfMeetings = if |argv| > 1 then
    this.extractor((argv(1)).to_int())
  else

Listing A.5: 7.Chameneos/List.enc

module List

import Std

read class Data[t] : Id
  val elem : t
  def init(elem : t) : unit
    this.elem = elem
  end
  def getValue() : t
    this.elem
  end
end

local class Link[t] : Id
  var data : Data[t]
  var next : Link[t]
  def init(elem : t, next : Link[t]) : unit
    let
      data = new Data(elem)
      in
        data.elem = elem
        this.data = data
        this.next = next
    end
  end
  def getData() : Data[t]
    this.data
  end
  def getNextLink() : Link[t]
    this.next
  end
  def show() : unit
    print("TBI")
    if this.next != null then
      this.next.show()
    else
      ()
  end
end
import Random

active class BigActor
  var id : int
  var numMessages : int
  var sinkActor : SinkActor
  var numPings : int
  var expPinger : int
  var neighbors : [BigActor]

A.5 Big

Listing A.6: 8.Big/Big.enc

local class List[t]: Id
  var first : Link[t]
  var size : int
  def init() : unit
    this.first = null : Link[t]
    this.size = 0
  end
  def prepend(elem : t) : unit
    let
      newFirst = new Link[t](elem, this.first)
    in
      this.first = newFirst
      this.size = this.size + 1
    end
  end
  def nth(var n : int) : Data[t]
    var cursor = this.first
    while n > 0 do
      cursor = cursor.getNextLink()
      n = n - 1
    end
    cursor.getData()
  end
  def pop() : Data[t]
    let
      head = this.first
    in
      this.first = this.first.getNextLink()
      this.size = this.size - 1
      head.getData()
    end
  end
  def show() : unit
    this.first.show()
  end
end
end
def init(id : int, numMessages : int, sinkActor : SinkActor) :
    unit
    this.id = id
    this.numMessages = numMessages
    this.sinkActor = sinkActor
    this.numPings = 0
    this.expPinger = -(1)
end
def ping(id : int) : unit
    ((this.neighbors)(id))! pong(this.id)
end
def pong(id : int) : unit
    if id != this.expPinger then
        print("ERROR: Expected: {} but received ping from {}\n", this.expPinger, id)
    end
    this.numPings = this.numPings + 1
    if this.numPings == this.numMessages then
        this.sinkActor! exit()
    else
        this! sendPing()
    end
end
def setNeighbors(neighbors : [BigActor]) : unit
    this.neighbors = neighbors
end
def sendPing() : unit
    let
        target = random(|this.neighbors|)
        targetActor = (this.neighbors)(target)
    in
        this.expPinger = target
        targetActor! ping(this.id)
    end
end
active class SinkActor
var numWorkers : int
var numMessages : int
def init(numWorkers : int) : unit
    this.numWorkers = numWorkers
    this.numMessages = 0
end
def exit() : unit
    this.numMessages = this.numMessages + 1
    if this.numMessages == this.numWorkers then
        print("Everything should be done now!\n")
    end
end
active class Main
def extractor(maybe : Maybe[int]) : int
    match maybe with
        case Nothing =>
            -(1)
        end
        case Just(i) =>
def main(argv : [String]) : unit
-- TODO: the original number of messages is 16 * 1024.
-- we had to reduce the size of numMessages because
-- it uses too much memory, and the process was killed
-- by OS before exiting. More information can be found
-- at
-- https://github.com/parapluu/ encore/issues/743
let
  numMessages = 1024 -- 16 * 1024
  numActors = if |argv| > 1 then
    this.extractor((argv(1)).to_int())
  else
    8 * 1024
  end
  sinkActor = new SinkActor(numActors)
  chunkSize = 1 * 1024
  chunks = if numActors >= chunkSize then
    numActors / chunkSize
  else
    1
  end
in
  var counter = 1
  while counter < chunks + 1 do
    let
      arraySize = if chunkSize * counter + 1 > numActors then
        if numActors > chunkSize then
          chunkSize + numActors % chunkSize
        else
          numActors
        end
      else
        chunkSize
      end
      bigActors = new [BigActor](arraySize)
    in
      repeat i <- |bigActors| do
        bigActors(i) = new BigActor(i, numMessages, sinkActor)
      end
      repeat i <- |bigActors| do
        (bigActors(i))!setNeighbors(bigActors)
      end
      repeat i <- |bigActors| do
        (bigActors(i))!pong(-(1))
      end
    end
    counter = counter + 1
  end
end
A.6 BndBuffer

Listing A.7: 11.BndBuffer/main.enc

```plaintext
import Manager

active class Main
def main() : unit
  let
    bufferSize = 50
    numProducers = 40
    numConsumers = 40
    numItemsPerProducer = 1000
    produce_cost = 25
    consume_cost = 25
  in
    new Manager(bufferSize, numProducers, numConsumers, numItemsPerProducer, produce_cost, consume_cost)
  end
end
end
```

Listing A.8: 11.BndBuffer/Consumer.enc

```plaintext
module Consumer
import Manager
import Random

fun math_log(x:real) : real
  EMBED (real) log(#(x)); END
end

fun math_abs(a:int) : int
  if a < 0 then
    0-a
  else
    a
  end
end

fun processItem(curTerm:real, cost:int) : real
  val max = 4000000000
  var res = curTerm
  if cost > 0 then
    repeat i <- cost do
      repeat j <- 100 do
        res = res + math_log(math_abs(random(max)) + 0.01)
      end
    end
  else
    res = res + math_log(math_abs(random(max)) + 0.01)
  end
  -- print ("Consumer is processing items")
  res
end
```
active class Consumer
val id : int
val manager : Manager
val consCost : int
var consItem : real

def init(id:int, manager:Manager, consCost:int) : unit
    this.id = id
    this.manager = manager
    this.consCost = consCost
    -- print("Consumer {} created\n", id)
end

def doConsume(data: real) : unit
    this.consItem = processItem(this.consItem + data, this.consCost)
end

def process(data: real) : unit
    this.doConsume(data)
    this.manager ! consumerAvailable(this)
end

Listing A.9: 11.BndBuffer/HackyQueue.enc

module HackyQueue

import Std

local class HackyQueue[t]: Id
val head : HackyQueueLink[t]
var tail : HackyQueueLink[t]
var size : int

def init(null_element : t) : unit
    this.head = new HackyQueueLink[t](null_element)
    this.tail = this.head
end

def append(element : t) : unit
    this.tail.next = new HackyQueueLink[t](element)
    this.tail = this.tail.next
    this.size = this.size + 1
end

def take() : t
    var dummy = this.head
    if dummy.next == null then
        dummy.element
    else
        this.size = this.size - 1
        val first = dummy.next
dummy.next = first.next
```python
if this.tail == first then
    this.tail = this.head
end
first.element
end

def size() : int
    this.size
end

def is_empty() : bool
    this.size == 0
end

subord class HackyQueueLink[t] : Id
    val element : t
    var next : HackyQueueLink[t]

def init(element : t) : unit
    this.element = element
end

Listing A.10: 11.BndBuffer/Manager.enc

module Manager
import HackyQueue
import Producer
import Consumer

active class Manager
    val adjusted_buffer_size : int
    val available_producers : HackyQueue[Producer]
    val available_consumers : HackyQueue[Consumer]
    val pending_data : HackyQueue[real]

def init(buffer_size: int, num_producers: int, num_consumers: int, items_per_producer: int, produce_cost:int, consume_cost: int) : unit
    this.adjusted_buffer_size = buffer_size - num_producers
    this.available_producers = new HackyQueue[Producer](null)
    this.available_consumers = new HackyQueue[Consumer](null)
    this.pending_data = new HackyQueue[real](-1)
    var producers = new [Producer](num_producers)
    repeat i <- num_producers do
        let p = new Producer(i, this, items_per_producer, produce_cost) in
            this.available_producers.append(p)
            producers(i) = p
    end
end
```

63
repeat i <- num_consumers do
    this.available_consumers.append(new Consumer(i, this, consume_cost))
end

for p <- producers do
    p!produce()
end
-- print("Hello, I am your new manager\n")
end

def dataItem(data : real, from : Producer) : unit
    if this.available_consumers.is_empty() then
        this.pending_data.append(data)
    else
        this.available_consumers.take()!process(data)
        -- print("Consumed {} litres\n", data)
    end
    if this.pending_data.size() >= this.adjusted_buffer_size then
        this.available_producers.append(from)
    else
        from!process()
    end
end

def consumerAvailable(consumer : Consumer) : unit
    if this.pending_data.is_empty() then
        this.available_consumers.append(consumer)
    else
        consumer!process(this.pending_data.take())
    end
    if not this.available_producers.is_empty() then
        this.available_producers.take()!process()
    end
end
end
end

Listing A.11: 11.BndBuffer/Producer.enc
this.manager = manager
this.numItemsToProduce = numItemsToProduce
this.prodCost = prodCost

-- print("Producer {} created\n", id)
end
def produce() : unit
    this.prodItem = processItem(this.prodItem, this.prodCost)
    this.manager ! dataItem(this.prodItem, this)
    this.itemsProduced = this.itemsProduced + 1
end
def process() : unit
    if this.itemsProduced <= this.numItemsToProduce then
        this.produce()
    end
end


-- This module could be removed once the new syntax is merged in
the development branch.
module Random

EMBED BODY
__thread unsigned seed;
END

-- Random number generator with thread-local seed
fun random(max:int) : int
    EMBED (int)
    if (seed == 0)
        {seed = (unsigned) time(NULL);
        }
    rand_r(&seed) % #{max};
END
e

-- Simple random number generator encapsulating a seed of its own
read class Random
    val seed:EMBED unsigned int END
-- Trace function required because of the embed type of seed
def Random_trace() : unit
    ()
end

e
-- Initialisation always with a seed in [0..2^32)
def init(seed:int) : unit
    assertFalse(seed < 0)
    assertFalse(seed > 4294967295) -- for want of an Encore 2^32
    operator
    EMBED (unit)
        _this->_enc__field_seed = (unsigned) #{seed};
-- Returns random % max where max [0..2^32)
def random(max:int) : int
    EMBED (int)
    rand_r((unsigned *) &(_this->enc_.field_seed)) % max;
END
end
end

module Util
import Random

fun math_log(x:real) : real
    EMBED (real)
    log(#x);
END
end

fun math_abs(a:int) : int
    if a < 0 then 0-a else a end
end

fun processItem(curTerm:real, cost:int) : real
    val max = 4000000000
    var res = curTerm
    if cost > 0 then
        repeat i <- cost do
            repeat j <- 100 do
                res = res + math_log(math_abs(random(max)) + 0.01)
            end
        end
    else
        res = res + math_log(math_abs(random(max)) + 0.01)
    end
res
end