

Exceptions for Algorithmic Skeletons

Mario Leyton[†] and Ludovic Henrio[‡] and José M. Piquer[§]

[†]NIC Labs, Universidad de Chile
Miraflores 222, Piso 14, 832-0198, Santiago, Chile.

`mleyton@niclabs.cl`

[‡]INRIA Sophia-Antipolis, , Université de Nice Sophia-Antipolis, CNRS - I3S
2004 Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France.

`First.Last@sophia.inria.fr`

[§]Departamento de Ciencias de la Computación, Universidad de Chile
Av. Blanco Encalada 2120, Santiago, Chile.

`jpiquer@dcc.uchile.cl`

Abstract. Algorithmic skeletons offer high-level abstractions for parallel programming based on recurrent parallelism patterns. Patterns can be combined and nested into more complex parallelism behaviors. Programmers fill the skeleton patterns with the functional (business) code, which transforms the generic skeleton into a specific application. However, when the functional code generates exceptions, programmers are exposed to implementation details of the skeleton library, breaking the high-level abstraction principle. Furthermore, related parallel activities must be stopped as the exception is raised. This paper describes how to handle exceptions in algorithmic skeletons without breaking the high-level abstractions of the programming model. We describe both the behavior of the framework in a formal way, and its implementation in Java: the Skandium Library.

Keywords: Algorithmic skeletons, exceptions, semantics

1 Introduction

Exceptions are the traditional way of handling programming errors which alter the normal execution flow. Many programming languages provide support for exception handling such as: Ada, C++, Eiffel, Java, Ocaml, Ruby, etc.

Algorithmic skeletons (*skeletons* for short) are a high-level programming model for parallel and distributed computing, introduced by Cole [7]. Skeletons take advantage of recurrent programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by nesting the basic ones.

To write an application, programmers must compose skeleton patterns and fill them with the sequential blocks specific to the application. The skeleton pattern implicitly defines the parallelization, distribution, orchestration, and composition aspects, while the functional code provides the application's functional aspects (i.e. business code).

The functional code, provided by users, is likely to encounter errors and generate exceptions. The open question we address in this paper is how exceptions, raised by a business code, interact with the surrounding skeleton pattern to alter (or not) the pattern’s normal execution flow. And in the worst case, how this exceptions are reported back to the parent skeleton or user, after related parallel activities are aborted.

In this paper we present an exception mechanism for an algorithmic skeleton library which blends in with the host programming language’s exception handling. Furthermore, the process of unwinding the stack does not reveal unnecessary low-level details of the skeleton library implementation, but is consistent with the high-level abstractions of the programming model.

This paper is organized as follows. Section 2 describes the related work. Section 3 provides a brief overview of the algorithmic skeleton programming model. Section 4 introduces the exception model. Section 5 shows how the exception model is implemented in the Skandium library, and Section 6 provides the conclusions.

2 Related Work

As a skeleton library we use Skandium [19,18], which is a multi-core reimplementation of Calcium [16], a ProActive [5] based algorithmic skeleton library. Skandium is mainly inspired by Lithium [1] and Muskel [9] frameworks developed at University of Pisa. In all of them, skeletons are provided to the programmer as a Java API. Our previous work has provided formalisms for algorithmic skeletons such as a type system for nestable parallelism patterns [6] and reduction semantics [16]. This work extends both previous formalisms.

QUAFF [10] is a recent skeleton library written in C++ and MPI. QUAFF relies on template-based meta-programming techniques to reduce runtime overheads and perform skeleton expansions and optimizations at compilation time. Skeletons can be nested and sequential functions are stateful. QUAFF takes advantage of C++ templates to generate, at compilation time, new C/MPI code. QUAFF is based on the CSP-model, where the skeleton program is described as a *process network* and production rules (single, serial, par, join) [11].

Formalisms in Skil [3] provide polymorphic skeletons in C. Skil was later reimplemented as the Muesli [14] skeleton library, but instead of a subset of the C language, skeletons are offered through C++. Contrary to Skil, Muesli supports nesting of task and data parallel skeletons [15] but is limited to P³L’s two tier approach [2].

Exceptions are a relevant subject on parallel programming models. For example in [4] exceptions for asynchronous method calls on active objects are introduced, and [12] provides formalisms for exception management in BSML, a functional parallel language for BSP. However, to the best of our knowledge, no previous work has focused on exception handling for algorithmic skeletons.

Regarding exception management in parallel and distributed programming, the exception handling approach proposed by Keen et al [13] for asynchronous method invocation presents similarities with our approach. In the sense that exceptions within the skeleton framework happen in an asynchronous context and we require a predefined handler to manage the exception. However, in our approach handlers are allowed to fail and we use the familiar `Future.get()` as a synchronization point to deliver the result or exception.

The exception handling approach used by JCilk [8] also presents a similarity with our approach. An exception raised by parallel a activity causes its siblings to abort. This yields simpler semantics, closer to what a sequential programmer would expect in Java; and also provides a mechanism to abort parallel activities in search type algorithms (such as branch and bound). However, the approach used in JCilk corresponds to a language extension while our approach is implemented as a library extension. Furthermore, our approach provides exceptions which do not break the high-level principle.

3 Algorithmic Skeletons in a Nutshell

In Skandium [19], skeletons are provided as a Java library. The library can nest task and data parallel skeletons in the following way:

$$\begin{aligned} \Delta ::= & \text{seq}(f_e, \langle h \rangle) \mid \text{farm}(\Delta, \langle h \rangle) \mid \text{pipe}(\Delta_1, \Delta_2, \langle h \rangle) \mid \text{while}(f_c, \Delta, \langle h \rangle) \mid \\ & \text{for}(i, \Delta, \langle h \rangle) \mid \text{if}(f_c, \Delta_{\text{true}}, \Delta_{\text{false}}, \langle h \rangle) \mid \text{map}(f_s, \Delta, f_c, \langle h \rangle) \mid \\ & \text{fork}(f_s, \{\Delta_i\}, f_m, \langle h \rangle) \mid \text{d\&c}(f_c, f_s, \Delta, f_m, \langle h \rangle) \end{aligned}$$

Each skeleton represents a different pattern of parallel computation. For each skeleton, an optional argument ($\langle \rangle$) has been introduced in this paper, which corresponds to the skeleton’s exception handler (h) described in detail in Section 4. In algorithmic skeletons all communication details are implicit for each pattern, hidden away from the programmer. The task parallel skeletons are: *seq* for wrapping execution functions; *farm* for task replication; *pipe* for staged computation; *while/for* for iteration; and *if* for conditional branching. The data parallel skeletons are: *map* for single instruction multiple data; *fork* which is like *map* but applies multiple instructions to multiple data; and *d&c* for divide and conquer.

The nested skeleton pattern (Δ) relies on sequential blocks of the application. These blocks provide the business logic and transform a general skeleton pattern into a specific application. We denominate these blocks *muscles*, as they provide the *real* (non-parallel) functionality of the program.

In Skandium, muscles come in four flavors:

Execution	$f_e : p \rightarrow r$
Split	$f_s : p \rightarrow [r_0, \dots, r_k]$
Merge	$f_m : [p_0, \dots, p_k] \rightarrow r$
Condition	$f_c : p \rightarrow \text{boolean}$

Here p ranges over parameters, i.e. values or object as represented in the host language (Java in our implementation). Additionally

$$r ::= p \mid e$$

where the result r can be either a new parameter p or an exception e ; and $[r_0, \dots, r_k]$ is a list of results.

For the skeleton language, muscles are black boxes invoked during the computation of the skeleton program. Multiple muscles may be executed either sequentially or in parallel with respect to each other, in accordance with the defined Δ . The result of a muscle is passed as a parameter to other muscle(s). When no further muscles need to be executed, the final result is delivered to the user.

4 Exceptions for Algorithmic Skeletons

Exceptions provide a useful mechanism to report errors and disrupt the normal program flow. Besides error communication, exceptions are also useful, for example, to stop computation once a result is found in recursive algorithms such as branch & bound. To illustrate the relevance of exceptions in algorithmic skeleton programming, consider the following simplified skeleton pseudocode:

```
Skeleton s = new Farm(new Pipe(stage1, stage2));
//...
public X stage1(P p) throws FileNotFoundException {
    FileInputStream fis = new FileInputStream(new File(p.filename));
}
```

Clearly muscles such as `stage1` can raise an exception (e.g. if `filename` is not found). However, for the skeleton pattern `Pipe` it is unclear whether the `FileNotFoundException` raised by `stage1` should be passed as a result to `stage2`, cancel computation altogether, or report the exception to the parent skeleton (`Farm`).

The model we propose contemplates three possible scenarios:

1. An exception is raised and caught inside a muscle. For the skeleton it is as if no exception was raised at all, and thus no action is required.
2. An exception is raised by a muscle or a sub-skeleton, and a matching handler is found and executed to produce the skeleton's result.
3. An exception is raised by a muscle or sub-skeleton, and no matching handler is found in the skeleton. The exception is raised to the parent skeleton.

4.1 Exception Semantics

A simplified version of the skeleton semantics is described, which extends a classical skeleton semantics that can be found in [17]. The principle of the operational semantics with exceptions is that muscle functions f are reduced to a result r which can be either a new parameter p or an exception e . Conceptually, when

an exception is raised by a muscle we want the surrounding skeleton to either handle the exception and produce a regular result, or raise the exception to the parent skeleton.

Consider the skeleton example: $farm(pipe(\Delta_1, \Delta_2, h))$ with the input p_0 . If $\Delta_1(p_0)$ reduces to a data p , then the evaluation continues normally, p is passed to Δ_2 and evaluated to p' :

$$farm(pipe(\Delta_1, \Delta_2, h))(p_0) \rightarrow \dots \rightarrow farm(pipe(p, \Delta_2, h)) \rightarrow farm(p') \rightarrow \dots$$

On the contrary, if Δ_1 is reduced to an exception e then:

$$farm(pipe(e, \Delta_2, h))(p_0) \rightarrow farm(h(e)) \rightarrow farm(r)$$

The *pipe* is reduced to $h(e)$ which returns a result r : either an exception e or a value p .

Extending the semantics for exceptions is tricky because nested skeletons are compiled into a sequence of lower-level instructions ($\Delta \rightarrow J$). Therefore the skeleton nesting must be remembered so that the instruction reduction semantics raise the exception to the parent skeleton's handler.

Let us introduce some new concepts and helper functions:

e is an an exception which represents an error during the computation. A concatenation $\tau \oplus e$ extends the trace of a given exception e by the creation context τ .

h is a programmer provided handler function. Given a parameter p and an exception e , this handler can be evaluated into a result: $h(p, e) \rightarrow r$. The result r can be a new parameter p' or a new exception e' if an error took place during the handler evaluation.

match is a function taking a handler h and an exception e as parameter. (h, e) returns *true* if a handler h can be applied to an exception e , and *false* otherwise.

\rightarrow and \Rightarrow arrows are used for local and global reductions respectively.

\uparrow is used to separate the evaluated skeleton from its exception handler.

An instruction J can reduce its parameter either to a result r or an exception e . For the later case, the handler h remembers the parameter p before J is reduced: p is both stored in the handler and used in J .

P-REMEMBER

$$J \uparrow h(\tau)(p) \rightarrow J(p) \uparrow h(\tau, p)$$

If the instruction finishes without raising an exception, then the handler is not invoked. Above, at the end of the pipe the handler h was discarded and $farm(p')$ is obtained.

FINISHED-OK

$$p \uparrow h(\tau, p') \Rightarrow p$$

On the contrary, if the instruction raises an exception then this exception can be transmitted, further instruction at the same level are discarded.

$$\begin{array}{c} \text{E-TRANSMIT} \\ J(e) \cdot \dots \cdot J_n \Rightarrow e \end{array}$$

If the result is an exception e and the handler h matches the exception, then the handler on the exception is invoked.

$$\begin{array}{c} \text{E-CATCH} \\ \frac{\text{match}(h, e)}{e \uparrow h(\tau, p) \Rightarrow h(\tau, p, e)} \end{array}$$

If the handler h does not match e , then we add a trace to the exception.

$$\begin{array}{c} \text{E-RAISE} \\ \frac{\neg \text{match}(h, e)}{e \uparrow h(\tau, p) \Rightarrow \tau \oplus e} \end{array}$$

In the scenario of data parallelism, if one of the subcomputations raises an exception:

$$\begin{array}{c} \text{CONQ-INST-REDUCE-WITH-EXCEPTION} \\ \text{conq}_I(f_c)(\Omega_1 \parallel \dots \parallel e_i \parallel \dots \parallel \Omega_n) \rightarrow \text{conq}_I(f_c)(e_i) \end{array}$$

Then the exception e_i is kept and the other parallel activities are discarded. For further details see Section 5.1.

4.2 Illustrative Example

Let us illustrate the semantics introduced in section 4.1 with a simple example showing some of the reduction rules. We consider the following skeleton

$$\Delta = \text{pipe}(if(f_b, \text{seq}(f_{pre}, h_p), \text{seq}(f_{id})), \text{seq}(f_t), h)$$

This skeleton acts in two phases. First, depending on a boolean condition on the received data (given by the muscle function f_b), a pre-treatment f_{pre} might be realized (or nothing if the condition is not verified, f_{id} is the identity muscle function); then, the main treatment, expressed by the function f_t is performed. The instruction corresponding to the preceding skeleton is the following:

$$\begin{array}{c} \Delta \rightarrow \text{pipe}_I(if_I(f_b, \text{seq}_I(f_{pre}) \uparrow h_p(\tau_p), \text{seq}_I(f_{id}) \uparrow h_\emptyset(\tau_1)) \\ \uparrow h_\emptyset(\tau_i), \text{seq}_I(f_t) \uparrow h_\emptyset(\tau_t)) \uparrow h(\tau) \end{array}$$

Where we introduce h_\emptyset as the empty handler, and τ_x for the locations of the different instructions. For example, τ is the creation point of the *pipe* instruction, and τ_i of the *if* instruction.

This instruction is evaluated as shown in Figure 1. Starting from an incoming data d_1 , we suppose that $f_b(d_1)$ is true. For reduction simplicity, we suppose that all the functions are stateless, allowing a parallel evaluation of the different steps. Reduced terms are underlined, and usage of `CONTEXT-HANDLER` and `CONTEXT-HANDLER-STACK` rules is implicit. (`INST-ARROW` allows reduction \rightarrow to be raised to the level of reduction \Rightarrow). The evaluation of the skeleton is shown in Figure 1, where \Rightarrow^* is the reflexive transitive closure of \Rightarrow . It involves an exception being raised and raised up to the top level handler.

$$\begin{aligned}
& pipe_I(if_I(f_b, seq_I(f_{pre}) \uparrow h_p(\tau_p), seq_I(f_{id}) \uparrow \emptyset(\tau_1)) \\
& \quad \uparrow \emptyset(\tau_i), seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau)([d_1]) \\
\Rightarrow^* & \frac{pipe_I(if_I(f_b, seq_I(f_{pre}) \uparrow h_p(\tau_p), seq_I(f_{id}) \uparrow \emptyset(\tau_1)) \\
& \quad \uparrow \emptyset(\tau_i), seq_I(f_t) \uparrow \emptyset(\tau_t))(d_1) \uparrow h(\tau, d_1)}{\uparrow \emptyset(\tau_i), seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)} && \text{REMEMBER} \\
\Rightarrow^* & \frac{if_I(f_b, seq_I(f_{pre}) \uparrow h_p(\tau_p), seq_I(f_{id}) \uparrow \emptyset(\tau_1)) \\
& \quad \uparrow \emptyset(\tau_i)(d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)}{\uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)} && \text{PIPE-REDUCTION-N} \\
\Rightarrow^* & \frac{if_I(f_b, seq_I(f_{pre}) \uparrow h_p(\tau_p), seq_I(f_{id}) \uparrow \emptyset(\tau_1))(d_1) \\
& \quad \uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)}{\uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)} && \text{REMEMBER} \\
\Rightarrow^* & \frac{seq_I(f_b)(d_1) \cdot choice_I(d_1, seq_I(f_{pre}) \uparrow h_p(\tau_p), seq_I(f_{id}) \uparrow \emptyset(\tau_1)) \\
& \quad \uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)}{\uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)} && \text{IF-INST} \\
\Rightarrow^* & \frac{true \cdot choice_I(d_1, seq_I(f_{pre}) \uparrow h_p(\tau_p), seq_I(f_{id}) \uparrow \emptyset(\tau_1)) \\
& \quad \uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)}{\uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)} && \text{SEQ} \\
\Rightarrow^* & \frac{seq_I(f_{pre}) \uparrow h_p(\tau_p)(d_1) \uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)}{\uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)} && \text{CHOICE} \\
\Rightarrow^* & \frac{e \uparrow h_p(\tau_p, d_1) \uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)}{\uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)} && \text{REMEMBER+SEQ} \\
\Rightarrow^* & \frac{\tau_i \oplus \tau_p \oplus e \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)}{\uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)} && \text{RAISE+FINISHED-OK} \\
\Rightarrow^* & \frac{\tau_i \oplus \tau_p \oplus e \uparrow h(\tau, d_1)}{\uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)} && \text{NEXT+PIPE} \\
\Rightarrow^* & h(d_1, \tau_i \oplus \tau_p \oplus e) && \text{CATCH+SEQ-INSTS}
\end{aligned}$$

Fig. 1. Exception Semantics Example

5 Exceptions in Skandium Library

Skandium is a Java based algorithmic skeleton library for high-level parallel programming of multi-core architectures. Skandium provides basic nestable parallelism patterns which can be composed to program more complex applications.

From a general perspective, parallelism or distribution of an application in Skandium is a producer/consumer problem. The shared buffer is a task queue, and the produced/consumed data are tasks, as shown in Figure 2.

A ready-queue stores ready tasks. Root-tasks are entered into the ready-queue by users, who provide the initial parameter and the skeleton program. The skeleton program undergoes a transformation process into an internal stack of instructions which harness the parallelism behavior of each skeleton (as detailed in Figure 2 of [17]).

Interpreter threads consume tasks from the ready-queue and compute their skeleton instruction stack. When the interpreters cannot compute a task any further, the task is either in finished or waiting state. If the task is in the finished state its result is delivered to the user. If the task is in the waiting state, then the task has generated new sub-tasks which are inserted into the ready-queue.

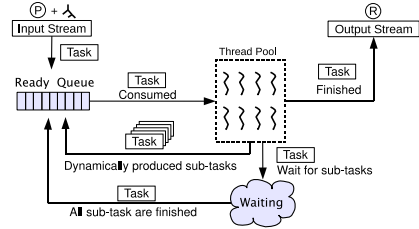


Fig. 2. Thread Pool Execution

```
1 // 1. Define the skeleton program
  Skeleton<Range, Range> sort = new DaC<Range, Range>(
3   new ShouldSplit(threshold),
   new SplitList(),
5   new Sort(),
   new MergeList(),
7   new HandleSortException());

9 // 2. Input parameters
  Future<Range> future = sort.input(new Range(...));
11
  // 3. Do something else here...
13
  // 4. Block for the results
15 try{
    Range result = future.get();
17 } catch(ExecutionException e){...}
```

Listing 1.1. Skandium Library Usage Output

Sub-tasks represent data parallelism for skeletons such as *map*, *fork*, *d&c*. A sub-task may in turn produce new sub-tasks. A task will exit the waiting state and be reinserted into the ready-queue when all of its sub-tasks are finished.

5.1 How Exceptions Cancel Parallelism

Exceptions disrupt the normal execution flow of a program. The generation and propagation of exceptions requires the cancellation of sibling parallel activities. The semantics introduced in Section 4.1 simply discards sibling parallel activities, as specified by the CONQ-INST-REDUCE-WITH-EXCEPTION rule. Thus, implementations are free to continue the execution of sibling parallel activities, and disregard their results, or apply a best effort to cancel the sibling parallel activities. The later approach is implemented in Skandium as it can be used to abort recursive searches, once a result is found.

In the case of the Skandium library, a task is cancelled as follows (this also applies to direct task cancellation by a user). If the task is in the ready-queue then it is removed. If the task is in execution, then it stopped as soon as possible: after the task's current instruction is finished, but before the next instruction begins. Finally, if the task is in waiting state, then each of its sub-tasks are cancelled (recursively).

A raised exception unwinds the task's execution stack until a handler is found and the computation can continue, or until the task's stack is empty. When the stack is empty, the exception is either returned to the user (for root-tasks) or passed to the parent-task (for sub-tasks). When a parent receives an exception from a sub-task all of the sub-task's siblings are cancelled, and the exception continues to unwind the parent's stack. Note that an exception propagation will not abort parallel activities from different root-tasks (task parallelism) as they do not have a common task ancestor.

```
1 class SplitList implements Split<Range, Range> throws ArrayIndexOutOfBoundsException{
3 @Override
  public Range[] split(Range r){
5     int i = partition(r.array, r.left, r.right);
7     Range[] intervals = {new Range(r.array, r.left, i-1), new Range(r.array, i+1, r.right)};
  return intervals;
9 }
```

Listing 1.2. Skandium Muscle Example

```
1 interface ExceptionHandler<P,R, E extends Exception>{
  public R handle(P p, E exception) throws Exception;
3 }
```

Listing 1.3. Exception Handler Interface

5.2 Skandium API with Exception

The code in Listing 1.1 shows how simple it is to interact with the Skandium API to input data and retrieve the result. Lines 1-7 show how a Divide and Conquer (DaC) skeleton is built using four muscle functions and an exception handler. Line 10 shows how new data is entered into the skeleton. In this case a new `Range(...)` contains an `array` and two indexes `left` and `right` which represent the minimum and maximum indexes to consider. The input yields a future which can be used to cancel the computation, query or block for results. Finally, line 16 blocks until the result is available or an exception is raised.

Listing 1.2 shows the definition of a Skandium muscle, which provides the functional (business) behavior to a skeleton. In the example, the `SplitList` muscle implements the `Split<P,R>` interface, and thus requires the implementation of the `R[] split(P p)` method. Additionally, the muscle may raise an `ArrayIndexOutOfBoundsException` when stepping outside of an array.

5.3 Exception Handler

An exception handler is a function $h : e \rightarrow r$ which transforms an exception into a regular result. All exception handlers must implement the `ExceptionHandler` interface shown in Listing 1.3.

The handler is in charge of transforming an exception raised by a sub-skeleton or muscle into a result. The objective is to have a chance to recover from an error and continue with the computation.

Listing 1.4 shows an example of an exception handler. If for some reason the `SplitList.split(...)` shown in Listing 1.2 raises an `ArrayIndexOutOfBoundsException`, then we use the handler to catch the exception and directly sort the array `Range` without subdividing.

```
1 class HandleSortException<Range, Range, ArrayIndexOutOfBoundsException> {
    public Range handle(Range r, Exception e){
3       Arrays.sort(r.array, r.left, r.right+1);
           return r;
5     }
}
```

Listing 1.4. Exception Handler Interface

```
Caused by: java.lang.Exception: Solve Test Exception
2 at examples.nqueens.Solve.execute(Solve.java:26)
  at examples.nqueens.Solve.execute(Solve.java:1)
4 at instructions.SeqInst.interpret(SeqInst.java:53)
  at system.Interpreter.interLoop(Interpreter.java:69)
6 at system.Interpreter.inter(Interpreter.java:163)
  at system.Task.run(Task.java:137)
```

Listing 1.5. Real Low-level Stack Trace

5.4 High-level Stack Trace

If the exception is raised outside of the Skandium library, i.e. when invoking `Future.get()` as shown in Listing 1.1, then this means that no handler was capable of catching the exception and the error is reported back to the programmer.

A stack-trace would normally have, for example, the form shown on Listing 1.5. This is the *real* stack-trace which corresponds to the actual Java stack-trace. In the example, line 2 generated the exception from inside the `Solve.execute(...)` muscle. This muscle was executed through a `SeqInst` instruction which in turn was called by the interpretation methods.

The problem with the *real* stack-trace, is that the lower-level implementation details are exposed to programmers. In this case the `SeqInst` instruction, thus breaking the high-level abstraction offered by algorithmic skeletons. Furthermore, the stack-trace is *flat* since there is no evidence of the skeleton nesting. In regular programming, this would be equivalent to printing only the name of the function which generated the exception, without actually printing the calling function. Furthermore, the function `Solve.execute()` could have been nested into more than one skeleton in the application, and it is impossible to know by inspecting the stack-trace from which of the nestings the exception was generated.

Therefore, we have introduced a *high-level stack-trace* which hides the internal interpretation details by not exposing instruction level elements and instead traces the skeleton nesting. The high-level stack-trace for the same example is shown in Listing 1.6. Lines 4-7 of the low-level stack-trace shown in Listing 1.5 are replaced by lines 4-6 in Listing 1.6. Thus it is evident that the error was generated by a `Solve` muscle nested inside a `DaC` which in turn was nested into a `Map` skeleton.

```
1 Caused by: java.lang.Exception: Solve Test Exception
  at examples.nqueens.Solve.execute(Solve.java:26)
3 at examples.nqueens.Solve.execute(Solve.java:1)
  at skeletons.Seq.<init>(DaC.java:68)
5 at skeletons.DaC.<init>(NQueens.java:53)
  at skeletons.Map.<init>(NQueens.java:60)
```

Listing 1.6. High-level Stack Trace

To produce a high-level stack-trace three steps are required in the library's implementation:

1. When a new skeleton object is created, it must remember its trace: class, method, file and line number. This corresponds to τ in Section 4.1.
2. When the skeleton object is transformed into an instruction, the trace must be copied to the instruction (Figure 1 of [17]).
3. When an exception is raised, an instruction unwinds the stack and adds its corresponding trace to the high-level stack-trace. This corresponds to the *E-RAISE* rule in Section 4.1.

6 Conclusions

We have presented an exception management model for algorithmic skeleton, in particular for the Java Skandium library which supports nestable parallelism patterns. The exception management has been formally specified, w.r.t. the library's operational semantics.

Exceptions can be raised and handled at each level of the skeleton nesting structure. Each skeleton can have handlers attached, specified by programmers through an API. The handlers are capable of catching errors and returning regular results, or raising the exception to be handled by the parent skeleton. Additionally, the raised exceptions are dynamically modified to reflect the nesting of skeleton patterns. Furthermore, no trace of lower-level library methods are exposed to programmers and exceptions do not break the abstraction level.

As future work we would like to apply this model to other skeleton (and related) frameworks. The ones which will benefit mostly from the proposed exception model are those that support nestable skeletons, asynchronous computations, and transformation of the skeleton patterns into lower-level modules that implement the parallelism behaviour.

References

1. Aldinucci, M., Danelutto, M., Teti, P.: An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems* 19(5), 611–626 (Jul 2003)
2. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P³L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience* 7(3), 225–255 (May 1995)

3. Botorog, G.H., Kuchen, H.: Efficient high-level parallel programming. *Theor. Comput. Sci.* 196(1-2), 71–107 (1998)
4. Caromel, D., Chazarain, G.: Robust exception handling in an asynchronous environment. In: Romanovsky, A., Dony, C., Knudsen, J., Tripathi, A. (eds.) *Proceedings of ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems*. Tech. Report No 05-050, Dept. of Computer Science, LIRMM, Montpellier-II Univ. July. France (2005)
5. Caromel, D., Delbé, C., di Costanzo, A., Leyton, M.: ProActive: an integrated platform for programming and running applications on Grids and P2P systems. *Computational Methods in Science and Technology* 12 (2006)
6. Caromel, D., Henrio, L., Leyton, M.: Type safe algorithmic skeletons. In: *Proceedings of the 16th Euromicro PDP*. pp. 45–53. IEEE CS Press, Toulouse, France (Feb 2008)
7. Cole, M.: *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA (1991)
8. Danaher, J.S., Lee, I.T.A., Leiserson, C.E.: Programming with exceptions in jcilck. *Sci. Comput. Program.* 63(2), 147–171 (2006)
9. Danelutto, M.: Qos in parallel programming through application managers. In: *Proceedings of the 13th Euromicro PDP*. pp. 282–289. IEEE Computer Society, Washington, DC, USA (2005)
10. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.T.: Quaff: efficient c++ design for parallel skeletons. *Parallel Computing* 32(7), 604–615 (2006)
11. Falcou, J., Sérot, J.: Formal semantics applied to the implementation of a skeleton-based parallel programming library. In: Joubert, G.R., Bischof, C., Peters, F.J., Lippert, T., Becker, M., Gibbon, P., Mohr, B. (eds.) *Parallel Computing: Architectures, Algorithms and Applications (Proc. of PARCO 2007, Julich, Germany)*. NIC, vol. 38, pp. 243–252. John von Neumann Institute for Computing, Germany (September 2007)
12. Gesbert, L., Loulergue, F.: Semantics of an exception mechanism for bulk synchronous parallel ml. In: *PDCAT '07: Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies*. pp. 201–208. IEEE Computer Society, Washington, DC, USA (2007)
13. Keen, A.W., Olsson, R.A.: Exception handling during asynchronous method invocation (research note). In: *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*. pp. 656–660. Springer-Verlag, London, UK (2002)
14. Kuchen, H.: A skeleton library. In: *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*. pp. 620–629. Springer-Verlag, London, UK (2002)
15. Kuchen, H., Cole, M.: The integration of task and data parallel skeletons. *Parallel Processing Letters* 12(2), 141–155 (2002)
16. Leyton, M.: *Advanced Features for Algorithmic Skeleton Programming*. Ph.D. thesis, Université de Nice-Sophia Antipolis (October 2008)
17. Leyton, M., Henrio, L., Piquer, J.M.: Operational semantics for algorithmic skeletons. Tech. rep., University of Chile (2010), to appear. <http://skandium.niclabs.cl/publications/semantics-2010.pdf>
18. Leyton, M., Piquer, J.M.: Skandium: Multi-core programming with algorithmic skeletons. In: *Proceedings of the 18th Euromicro PDP*. IEEE CS Press, Pisa, Italy (Feb 2010), to appear.
19. Skandium: <http://skandium.niclabs.cl/>