Exceptions for Algorithmic Skeletons

Mario Leyton, Ludovic Henrio, José Piquer

To cite this version:

Mario Leyton, Ludovic Henrio, José Piquer. Exceptions for Algorithmic Skeletons. 16th Int. European Conference on Parallel and Distributed Computing (Euro-Par 2010), 2010, Italy. hal-00486108

HAL Id: hal-00486108

https://hal.archives-ouvertes.fr/hal-00486108

Submitted on 25 May 2010

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 generate exceptions, this exposes the programmer to 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

Algorithmic skeletons (*skeletons* for short) is a high-level programming model for parallel and distributed computing, introduced by Cole [1]. 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 are addressing in this paper is how the exceptions raised by a functional 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 user, after aborting related parallel activities.

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.

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 de conclusions.

2 Related Work

As a skeleton library we use Skandium [2,3], which is a multi-core reimplementation of Calcium [4], a ProActive [5] based algorithmic skeleton library. Skandium is mainly inspired by Lithium [6] and Muskel [7] 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 [8] and reduction semantics [9]. 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 [12] provide polymorphic skeletons in C. Skil was later reimplemented as the Muesli [13] 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 [14] but is limited to P^{3}L's two tier approach [15].

Exceptions are a relevant subject on parallel programming models. For example in [16] exceptions for asynchronous method calls on active objects are introduced, and [17] 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 [18] 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 `Future.get()` as a synchronization point to deliver the result or exception.

The exception handling approach used by JCilk [19] 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 the our approach is implemented as a library extension. Furthermore, our approach provides exception which do not break the high-level principle.

3 Algorithmic Skeletons in a Nutshell

In Skandium [2], skeletons are provided as a Java library. The libraryw can nest task and data parallel skeletons in the following way:

$$\begin{aligned}\triangle ::= &\ \mathrm{seq}(f_e, \langle h\rangle) \mid \mathrm{farm}(\triangle, \langle h\rangle) \mid \mathrm{pipe}(\triangle_1, \triangle_2, \langle h\rangle) \mid \mathrm{while}(f_c, \triangle, \langle h\rangle) \mid \\ &\ \mathrm{for}(i, \triangle, \langle h\rangle) \mid \mathrm{if}(f_c, \triangle_{true}, \triangle_{false}, \langle h\rangle) \mid \mathrm{map}(f_s, \triangle, f_c, \langle h\rangle) \mid \\ &\ \mathrm{fork}(f_s, \{\triangle_i\}, f_m, \langle h\rangle) \mid \mathrm{d\&c}(f_c, f_s, \triangle, f_m, \langle h\rangle)\end{aligned}$$

Each skeleton represents a different pattern of parallel computation. An optional argument ($\langle\rangle$) has been introduced, for each skeleton, 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.

3.1 Muscle (Sequential) Blocks

The nested skeleton pattern ($\triangle$) 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 application.

In Skandium, muscles come in four flavors:

Execution	$f_e : p \rightarrow r$
Split	$f_s : p \rightarrow [r_0, ..., r_k]$
Merge	$f_m : [p_0, ..., p_k] \rightarrow r$
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

$$r ::= p \quad | \quad e$$

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

3.2 Skeleton's Semantics

Out operational semantics for Algorithmic Skeletons is detailed in Appendix A and illustrated by the example in Section 4.2. It operates as follows. A skeleton program $\triangle$ defined by the programmer is transformed into a lower level representation formed of instructions, with its corresponding muscle functions and exception handlers. Transforming high-level skeletons to lower-level constructs is a standard methodology in many Skeleton frameworks such as P3L, Lithium, Muesli, QUAFF, etc.

These lower-level instructions provide the actual parallelization semantics, and are grouped into stacks. When data parallelism is encountered, such as for map, fork, and d&c skeletons, a $DATA - ||$ rule creates new stacks with the instructions to be computed in parallel. The reduction of a skeleton consists in evaluating a set of stacks: the first instruction of each stack is evaluated until a result is obtained, then the result is passed to the next instruction of the task. When the stacks are finished, the results are merged back from the parallel activities with the CONQ-INST-REDUCE rule. Task parallelism can also be achieved ($TASK - ||$ rule) by computing the same instruction concurrently for different (unrelated) data.

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 computaton once a result is found in recursive algorithms such as branch & bound.

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

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 example: $farm(pipe(\triangle_1, \triangle_1, h))$. If $\triangle_1$ is reduced to an exception e then:

$$farm(pipe(e, \triangle_2, h)) \rightarrow farm(h(e)) \rightarrow farm(r)$$

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

The challenge is that nested skeletons are transformed into a sequence of lower-level instructions ($\triangle \twoheadrightarrow J$). 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:

An exception e 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.

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

If the instruction finishes normally, i.e. without raising an exception, then the handler is not invoked.

FINISHED-OK
$$p \uparrow h(\tau, p') \Rightarrow p$$

On the contrary, if the instruction raises an exception then this exception can be transmitted:

E-TRANSMIT
$$J(e) \cdot \ldots \cdot J_n \Rightarrow e$$

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

E-CATCH
$$\frac{match(h, e)}{e \uparrow h(\tau, p) \Rightarrow h(\tau, p, e)}$$

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

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

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

CONQ-INST-REDUCE-WITH-EXCEPTION
$$conq_I(f_c)(\Omega_1 \| \ldots \| e_i \| \ldots \| \Omega_n) \rightarrow conq_I(f_c)(e_i)$$

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 presented above on a simple example showing some of the reduction rules. We consider the following skeleton

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

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:

$$\triangle \twoheadrightarrow pipe_I(if_I(f_b, seq_I(f_{pre}) \uparrow h_p(\tau_p), seq_I(f_{id}) \uparrow h_\emptyset(\tau_1)) \\ \uparrow h_\emptyset(\tau_i), seq_I(f_t) \uparrow h_\emptyset(\tau_t)) \uparrow h(\tau)$$

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

According to the semantics defined above, 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. 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 raised and raised up to the top level handler.

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

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.

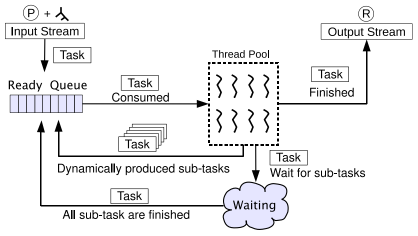

Fig. 2. Thread Pool Execution

From a general perspective, parallelism or distribution of an application in Skandium is a producer/consumer problem. Where 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 3 of Appendix A).

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.

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

// 2. Input parameters
Future<Range> future = sort.input(new Range(...));

// 3. Do something else here...

// 4. Block for the results
try{
 Range result = future.get();
} 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-taks. 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 Skanidum 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 tasks 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 aborted (recursively).

When an exception is raised, it 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 aborted, 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.

```java
class SplitList implements Split<Range, Range> throws ArrayIndexOutOfBoundsException{

@Override
public Range[] split(Range r){

  int i = partition(r.array, r.left, r.right);
  Range[] intervals ={new Range(r.array, r.left, i-1), new Range(r.array, i+1, r.right)};
  return intervals;
}
```

Listing 1.2. Skandium Muscle Example

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

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, lines 16-20 block 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 `ArrayIndexOutOf BoundException`, then we use the handler to catch the exception and directly sort the array `Range` without subdividing.

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

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, ie 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 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.

```
Caused by: java.lang.Exception: Solve Test Exception
at examples.nqueens.Solve.execute(Solve.java:26)
at examples.nqueens.Solve.execute(Solve.java:1)
at skeletons.Seq.<init>(DaC.java:68)
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 3 of Appendix A).
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

This paper has presented an exception management model for algorithmic skeleton, in particular for the Java Skandium Library which supports nestable parallelism patterns. The whole library, including exception management, has been formally specified, and the operational semantics for dealing with exception was also presented.

In our model, exceptions can be raised and handled at each level of the skeleton nesting structure. Each skeleton can have handlers attached, specified by programmers through and 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 excceptions do not break the abstraction level.

References

1. Murray Cole. *Algorithmic skeletons: structured management of parallel computation.* MIT Press, Cambridge, MA, USA, 1991.
2. Skandium. `http://skandium.niclabs.cl/`.
3. Mario Leyton and José M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Proceedings of the 18th Euromicro PDP*, Pisa, Italy, February 2010. IEEE CS Press. To appear.
4. Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In *13th International Euro-Par Conference: Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 2007.

5. Denis Caromel, Christian Delbé, Alexandre di Costanzo, and Mario Leyton. ProActive: an integrated platform for programming and running applications on Grids and P2P systems. *Computational Methods in Science and Technology*, 12, 2006.
6. Marco Aldinucci, Marco Danelutto, and Paolo Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.
7. Marco Danelutto. Qos in parallel programming through application managers. In *Proceedings of the 13th Euromicro PDP*, pages 282–289, Washington, DC, USA, 2005. IEEE Computer Society.
8. Denis Caromel, Ludovic Henrio, and Mario Leyton. Type safe algorithmic skeletons. In *Proceedings of the 16th Euromicro PDP*, pages 45–53, Toulouse, France, February 2008. IEEE CS Press.
9. Mario Leyton. *Advanced Features for Algorithmic Skeleton Programming.* PhD thesis, Université de Nice-Sophia Antipolis, October 2008.
10. J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. Quaff: efficient c++ design for parallel skeletons. *Parallel Computing*, 32(7):604–615, 2006.
11. Joel Falcou and Jocelyn Sérot. Formal semantics applied to the implementation of a skeleton-based parallel programming library. In G. R. Joubert, C. Bischof, F. J. Peters, T. Lippert, M. Bcker, P. Gibbon, and B. Mohr, editors, *Parallel Computing: Architectures, Algorithms and Applications (Proc. of PARCO 2007, Julich, Germany)*, volume 38 of *NIC*, pages 243–252, Germany, September 2007. John von Neumann Institute for Computing.
12. G. H. Botorog and H. Kuchen. Efficient high-level parallel programming. *Theor. Comput. Sci.*, 196(1-2):71–107, 1998.
13. Herbert Kuchen. A skeleton library. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 620–629, London, UK, 2002. Springer-Verlag.
14. Herbert Kuchen and Murray Cole. The integration of task and data parallel skeletons. *Parallel Processing Letters*, 12(2):141–155, 2002.
15. Bruno Bacci, Marco Danelutto, Salvatore Orlando, Sussana Pelagatti, and Marco Vanneschi. P^{3}L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, May 1995.
16. D. Caromel and G. Chazarain. Robust exception handling in an asynchronous environment. In A. Romanovsky, C. Dony, JL. Knudsen, and A. Tripathi, editors, *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.
17. Louis Gesbert and Frederic Loulergue. 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*, pages 201–208, Washington, DC, USA, 2007. IEEE Computer Society.
18. Aaron W. Keen and Ronald A. Olsson. Exception handling during asynchronous method invocation (research note). In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 656–660, London, UK, 2002. Springer-Verlag.
19. John S. Danaher, I.-Ting Angelina Lee, and Charles E. Leiserson. Programming with exceptions in jcilk. *Sci. Comput. Program.*, 63(2):147–171, 2006.

A Appendix: Operational Semantics for Algorithmic Skeletons

Notation. The semantics in this section distinguish three types of reductions. The "$\Rightarrow$" arrow used for global reductions where the context of the execution (e.g. other parallel processes) is explicitly expressed; the "$\rightarrow$" arrow used for local reductions occurring in a single process, it only focuses on local reductions of instructions that do not involve communications between processes; and the "$\twoheadrightarrow$" arrow used for transformations between languages (i.e. compilation).

A.1 The Instruction Language

Once the skeleton program is defined, it is transformed into a lower-level internal representation, hidden from programmers, in charge of exploiting the parallelism patterns. We call these internal representation instructions:

$$\begin{aligned} I ::= & id_I(f) \mid seq_I(f) \mid pipe_I(\{J\}) \mid if_I(f_c, J_{true}, J_{false}) \mid for_I(i, J) \mid while_I(f_c, J) \mid \\ & map_I(f_s, J, f_m) \mid fork_I(f_s, \{J\}, f_m) \mid d\&c_I(f_s, f_c, J, f_m) \mid \\ & choice_I(p, J_{true}, J_{false}) \mid div_I(\{J\}, f_c) \mid conq_I(f_m) \end{aligned}$$

$J ::= I \mid J \uparrow h(\tau)$

Where J is an instruction that allows for an exception handler $h(\tau)$. Several instructions can be concatenated into a stack S as:

S-DEF
$$S ::= r \cdot J_1 \cdot \ldots \cdot J_n \quad | \quad J(r) \cdot J_1 \cdot \ldots \cdot J_n$$

A stack my allow for an exception handler, which is ready to catch raised exceptions:

H-DEF
$$H ::= S \quad | \quad H \uparrow h(\tau, p) \quad | \quad H_1 \cdot \ldots \cdot H_n$$

Several H can exist in parallel ($\|$), and Ω ranges over all parallel compositions:

CONTEXT-DEF
$$\Omega ::= H \quad | \quad r \quad | \quad \Omega \parallel \Omega$$

Parallel activities are commutative, as the order in which stacks can be computed in parallel is irrelevant:

COMMUTATIVITY
$$\Omega \| \Omega' \equiv \Omega' \| \Omega$$

Also, parallel activities can progress:

$$\text{H-PROGRESS}\quad \frac{H \Rightarrow H'}{H \| \Omega \Rightarrow H' \| \Omega} \qquad \text{CONTEXT-HANDLER}\quad \frac{H \Rightarrow H'}{H \uparrow h(\tau, p) \Rightarrow H' \uparrow h(\tau, p)}$$

$$\text{CONTEXT-HANDLER-STACK}\quad \frac{H \Rightarrow H'}{H \cdot H_2 \cdot \ldots \cdot H_n \Rightarrow H' \cdot H_2 \cdot \ldots \cdot H_n}$$

When a parameter is delivered it is processed by the next instructoin J as follows:

$$\text{NEXT}\quad r \cdot J_1 \cdot \ldots \cdot J_n \Rightarrow J_1(r) \cdot \ldots \cdot J_n$$

A.2 Transformation Rules

Skeletons are transformed ($\triangle \twoheadrightarrow J$) into instructions as shown in Figure 3. For example, the *PIPE-TRANS* says that a *pipe* skeleton composed of a list of skeletons $[\triangle_1, ..., \triangle_k]$ and an exception handler h is transformed into a $pipe_I$ instruction composed of a list of instructions $[J_1, ..., J_k]$ and a handler $h(\tau)$, if each skeleton $\triangle_i$ can be transformed into a instructions J_i.

A.3 Reduction Rules

Non-atomicity is important to allow concurrent executions. For simplicity, the reduction rules presented in Figure 4 base all concurrent executions on the non-atomicity of seq_I as it is used to wrap the evaluation of muscles.

Therefore, for non-atomic execution of seq_I we define the following rules:

$$\text{NON-ATOMIC-SEQ-INST}\quad \frac{f(p) \rightarrow f'(p')}{seq_I(f)(p) \rightarrow seq_I(f')(p')} \qquad \text{RETURN-VALUE-SEQ-INST}\quad \frac{f(p) \rightarrow r}{seq_I(f)(p) \rightarrow r}$$

The **non-atomic-seq-inst** rule states that if a function $f(p)$ can be evaluated to an intermediate state $f'(p')$, then the $seq_I(f)(p)$ instruction can also be evaluated to an intermediate state $seq_I(f')(p')$.

If a skeleton receives multiple parameters, and the instructions of a stack are stateless, then they can be parallelized as follows:

$$\text{TASK-}\|\quad \frac{}{J[p_1, \ldots, p_m] \rightarrow J(p_1) \| \ldots \| J(p_m)}$$

This reduction is what we call task parallelism. The stack S is copied m times, and each copy is applied to one of the parameters.

On the other hand, data parallelism is expressed with the div_I and $conq_I$ instructions. The div_I instruction is reduced as follows:

SEQ-TRANS

$$seq(f,h) \twoheadrightarrow seq_I(f) \uparrow h(\tau)$$

FARM-TRANS

$$\frac{\triangle \twoheadrightarrow J}{farm(\triangle,h) \twoheadrightarrow J \uparrow h(\tau)}$$

PIPE-TRANS

$$\frac{\triangle_i \twoheadrightarrow J_i \quad \forall i/0 < i \leq k}{pipe([\triangle_1,...,\triangle_k],h) \twoheadrightarrow pipe_I(\underbrace{[J_1,...,J_k]}_{\text{length } k}) \uparrow h(\tau)}$$

WHILE-TRANS

$$\frac{\triangle \twoheadrightarrow J}{while(f_c,\triangle,h) \twoheadrightarrow while_I(f_c,J) \uparrow h(\tau)}$$

IF-TRANS

$$\frac{\triangle_{true} \twoheadrightarrow J_{true} \quad \triangle_{false} \twoheadrightarrow J_{false}}{if(f_c,\triangle_{true},\triangle_{false},h) \twoheadrightarrow if_I(f_c,J_{true},J_{false}) \uparrow h(\tau)}$$

FOR-TRANS

$$\frac{\triangle \twoheadrightarrow J}{for(n,\triangle,h) \twoheadrightarrow for_I(n,J) \uparrow h(\tau)}$$

MAP-TRANS

$$\frac{\triangle \twoheadrightarrow J}{map(f_s,\triangle,f_m,h) \twoheadrightarrow map_I(f_s,J,f_m) \uparrow h(\tau)}$$

FORK-TRANS

$$\frac{\forall \triangle_i \in \{\triangle\} \quad \triangle_i \twoheadrightarrow J_i}{fork(f_s,\{\triangle\},f_m,h) \twoheadrightarrow fork_I(f_s,\{S\},f_m) \uparrow h(\tau)}$$

D&C-TRANS

$$\frac{\triangle \twoheadrightarrow J}{d\&c(f_c,f_s,\triangle,f_n,h) \twoheadrightarrow d\&c_I(f_c,f_s,J,f_m) \uparrow h(\tau)}$$

Fig. 3. Transformation Rules

DATA-$\|$

$$\frac{\{J\} = [J_1, .., J_n]}{div_I(\{J\}, f_c)([p_1, ..., p_n]) \rightarrow \qquad conq_I(f_c)(J_1(p_1)\| \ldots \|J_n(p_n))}$$

The list of parameters $[p_1, ..., p_n]$ is spread over the list of instructions $\{J\}$ and applied as $J_i(p_i)$. Data parallelism is achieved since each instruction can be computed in parallel with the others.

The progress of the parallel activities is reflected in the *conq-inst-progress* rule. When the evaluation of the parallel activities is concluded $(r_1\| \ldots \|r_n)$, the results are passed as parameters to the $conq_I$ instruction and reduced by the *conq-inst-reduce* rule:

CONQ-INST-PROGRESS

$$\frac{\Omega \Rightarrow \Omega'}{conq_I(f_c)(\Omega) \rightarrow conq_I(f_c)(\Omega')}$$

CONQ-INST-REDUCE

$$conq_I(f_c)(r_1\| \ldots \|r_n) \rightarrow seq(f_c)([r_1, ..., r_n])$$

Note that values (or results) are totally reduced terms that could also be called normal forms.

PIPE-REDUCTION-N
$$\frac{k \geq 1}{pipe_I([J_1, ..., J_k])(p) \rightarrow J_1(p) \cdot pipe_I([J_2, ..., J_k])}$$

PIPE-REDUCTION-0
$$pipe_I([])(p) \rightarrow p$$

IF-INST
$$if_I(f_c, J_{\text{true}}, J_{\text{false}})(p) \rightarrow seq_I(f_c)(p) \cdot choice(p, J_{\text{true}}, J_{\text{false}})$$

ID-INST
$$id_I(p) \rightarrow p$$

CHOICE-INST-TRUE
$$choice_I(p, J_{\text{true}}, J_{\text{false}})(\text{true}) \rightarrow J_{\text{true}}(p)$$

CHOICE-INST-FALSE
$$choice_I(p, J_{\text{true}}, J_{\text{false}})(\text{false}) \rightarrow J_{\text{false}}(p)$$

WHILE-INST
$$while_I(f_c, J)(p) \rightarrow if_I(f_c, J \cdot while_I(f_c, J), id_I)(p)$$

FOR-INST-N
$$\frac{n > 0}{for_I(n, J)(p) \rightarrow J(p) \cdot for_I(n-1, J)}$$

FOR-INST-0
$$for_I(0, J)(p) \rightarrow p$$

MAP-INST
$$map_I(f_s, J, f_m)(p) \rightarrow seq_I(f_s)(p) \cdot div_I(\overbrace{[J, \ldots, J]}^{k \text{ times}}, f_m)$$

FORK-INST
$$fork_I(f_s, \{J\}, f_m)(p) \rightarrow seq_I(f_s)(p) \cdot div_I(\{J\}, f_m)$$

D&C-INST
$$\frac{I = d\&c_I(f_c, f_s, J, f_m)}{I(p) \rightarrow if_I(f_c, seq_I(f_s) \cdot div_I([I, \ldots, I], f_m), J)(p)}$$

INST-ARROW
$$\frac{I(p) \rightarrow J(r) \cdot J_2 \cdot \ldots \cdot J_n}{I(p) \Rightarrow J(r) \cdot J_2 \cdot \ldots \cdot J_n}$$

NON-ATOMIC-SEQ-INST
$$\frac{f(p) \rightarrow f'(p')}{seq_I(f)(p) \rightarrow seq_I(f')(p')}$$

RETURN-VALUE-SEQ-INST
$$\frac{f(p) \rightarrow r}{seq_I(f)(p) \rightarrow r}$$

TASK-$\|$
$$\frac{}{J[p_1, \ldots, p_m] \rightarrow J(p_1) \| \ldots \| J(p_m)}$$

DATA-$\|$
$$\frac{\{J\} = [J_1, .., J_n]}{div_I(\{J\}, f_c)([p_1, ..., p_n]) \rightarrow \qquad conq_I(f_c)(J_1(p_1) \| \ldots \| J_n(p_n))}$$

CONQ-INST-PROGRESS
$$\frac{\Omega \Rightarrow \Omega'}{conq_I(f_c)(\Omega) \rightarrow conq_I(f_c)(\Omega')}$$

CONQ-INST-REDUCE
$$conq_I(f_c)(r_1 \| \ldots \| r_n) \rightarrow seq(f_c)([r_1, ..., r_n])$$

Fig. 4. Reduction Rules