Typing and Exceptions for Algorithmic Skeletons

Ludovic Henrio
INRIA Sophia-Antipolis, CNRS,
Univ of Nice Sophia-Antipolis, France

Realised in collaboration with *Mario Leyton*,
NIC Chile Research Labs

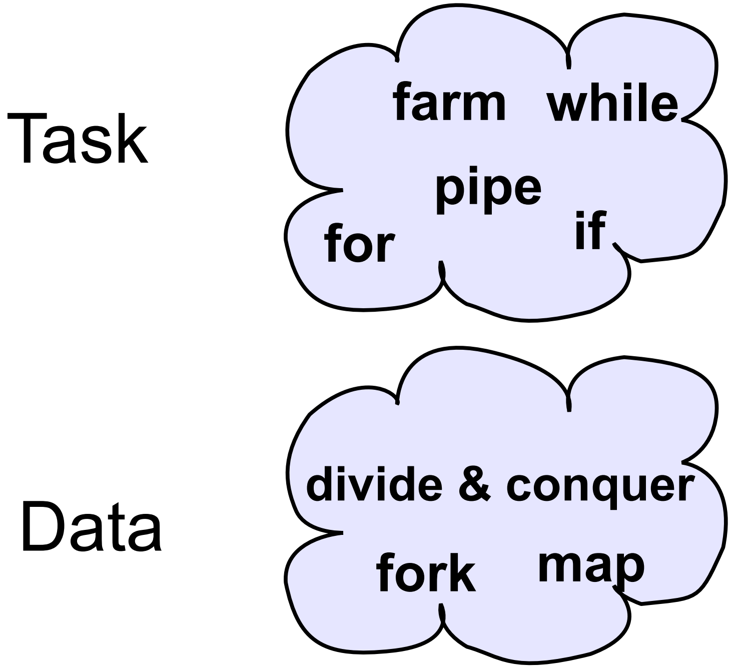
Agenda

- **Introduction to Algorithmic Skeletons**
- Our Skeletons and Libraries
- Typing Skeletons
- Exceptions
- Semantics for Skeletons and Exceptions
- Conclusion and Future Work

What are Algorithmic Skeletons?

- High Level Programming Model [*Cole89*]
- Hides the complexity of parallel/distributed programming
- Exploits nestable parallelism patterns

Parallelism Patterns



Task

farm while

pipe

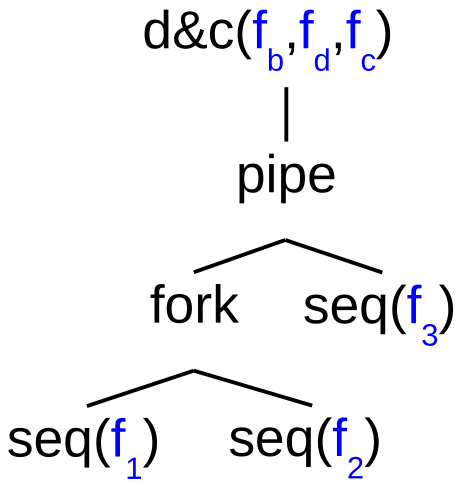
for if

Data

divide & conquer

fork map

BLAST
Skeleton Program



d&c(f_b,f_d,f_c)

pipe

fork seq(f_3)

seq(f_1) seq(f_2)

What are Algorithmic Skeletons?

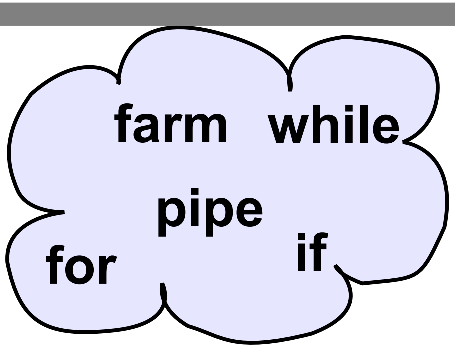
```
public boolean condition(BlastParams param)
{

    File file = param.dbFile;

    return file.length() > param.maxDBSize;
}
```

mming

T

Program

Task

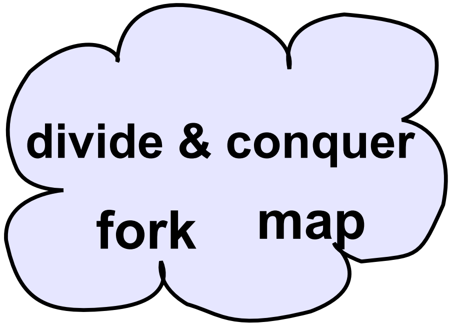


farm while

pipe

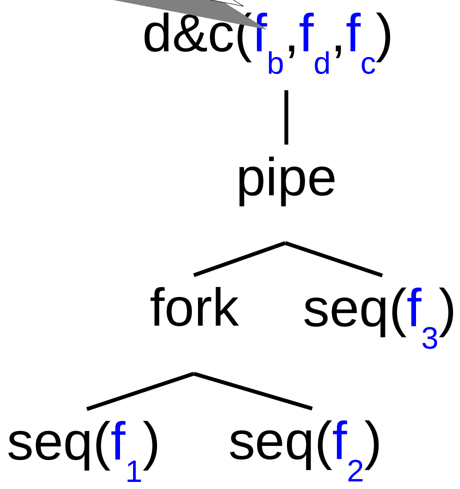
for if

Data



divide & conquer

fork map



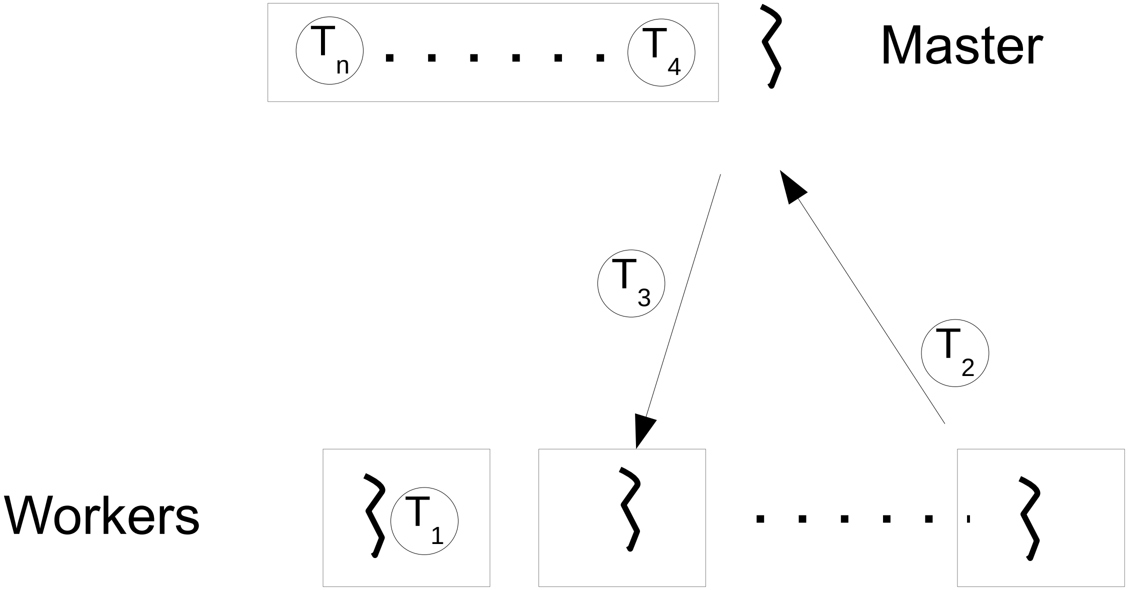
d&c(f_b,f_d,f_c)

pipe

fork seq(f_3)

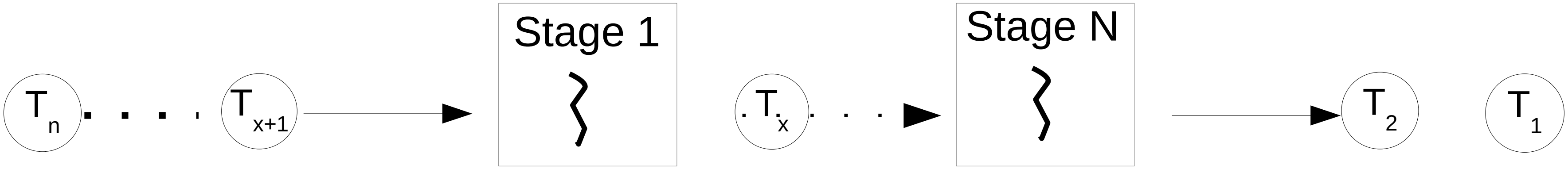
seq(f_1) seq(f_2)

The Farm
(a.k.a. Master-Slave)



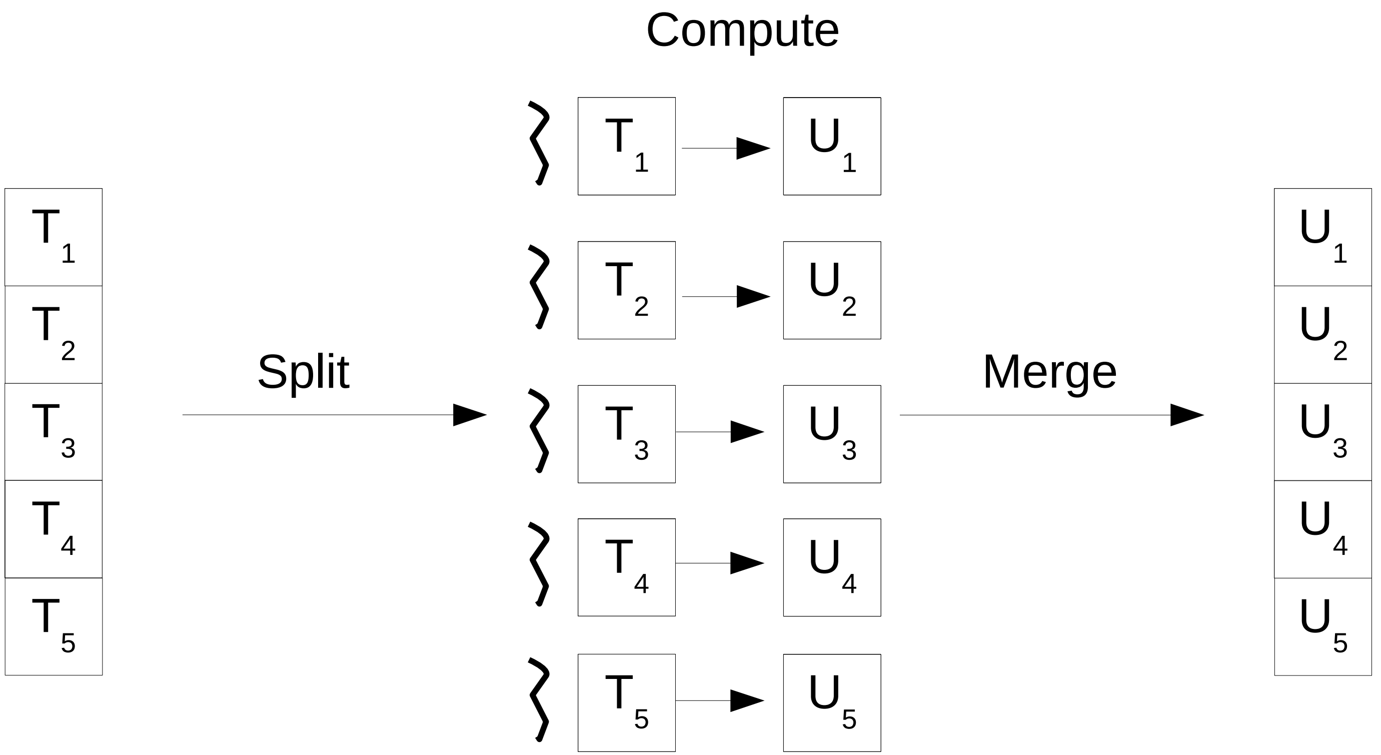
- **The** skeleton for embarrassingly parallel applications.
- Features: dispatching, load balancing, fault tolerance
- Note that each **worker** can be another skeleton

The Pipe



- Use cases:
  - Image manipulation
  - Preliminary preparation
- Note that each **stage** can be another skeleton

The Map



- Note that the **split**, **compute** and **merge** may be other skeletons

Agenda

- Introduction to Algorithmic Skeletons
- **Our Skeletons and Libraries**
- Typing Skeletons
- Exceptions
- Semantics for Skeletons and Exceptions
- Conclusion and Future Work

Skandium Library / Calcium Library

http://skandium.niclabs.cl
http://proactive.objectweb.org

- Java library, Open Source
- Nestable Task and Data parallel Skeletons
- Type Safe Skeleton API
- Multi-core parallelism in Skandium / Distribution in Calcium (Active Objects)
- Exception management in Skandium
- Examples Bundle:
  - QuickSort, NQueens, Pi Decimals.

Our Algorithmic skeletons

Δ ::= seq(f_e) | farm(Δ) | pipe(Δ_1, Δ_2)

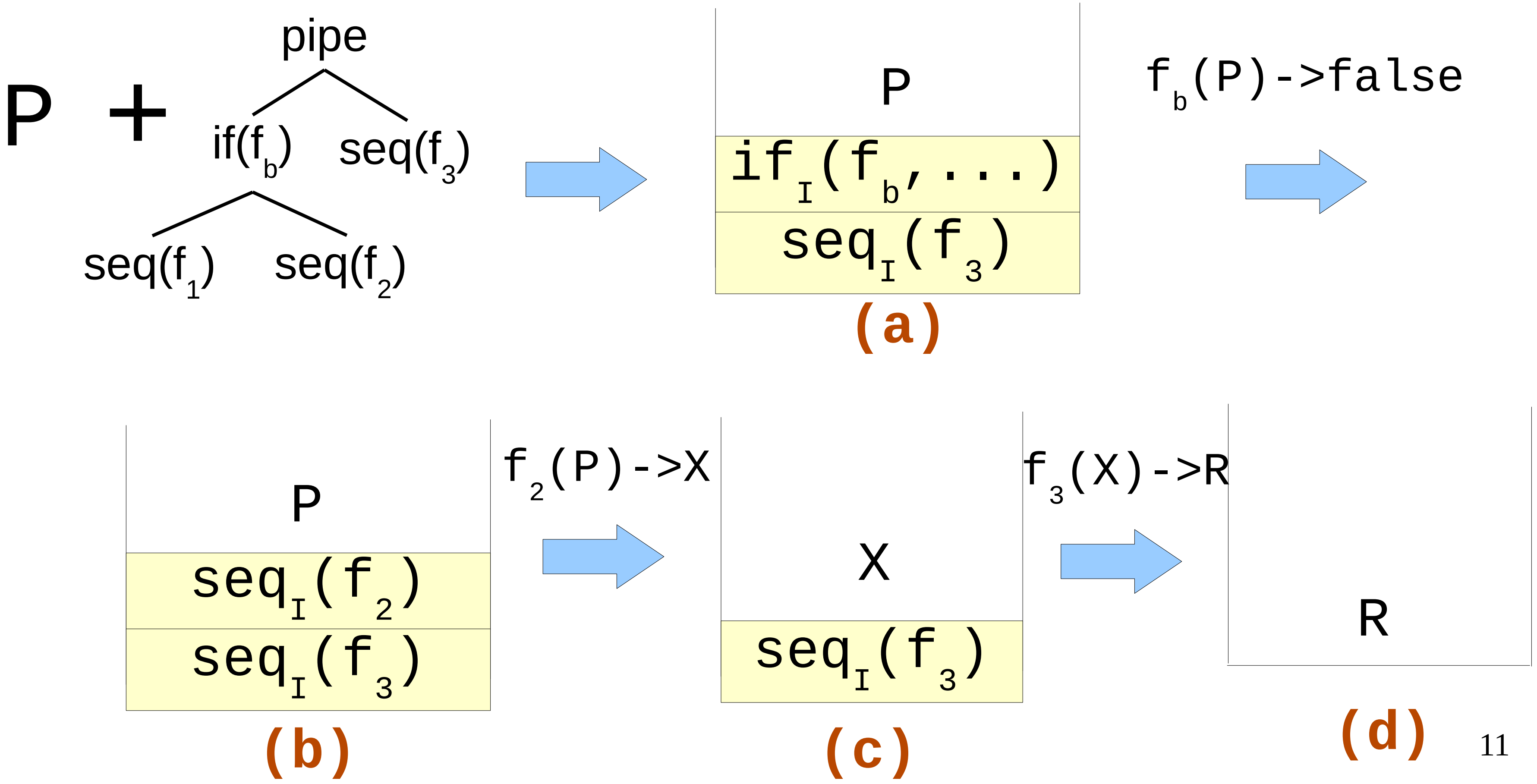
while(f_c, Δ) | for(i, Δ) | if(f_c, Δ_{true} , Δ_{false})

map(f_s, Δ, f_m) | fork(f_s,{Δ_i}, f_m) | d&c(f_s, f_c, Δ, f_m)

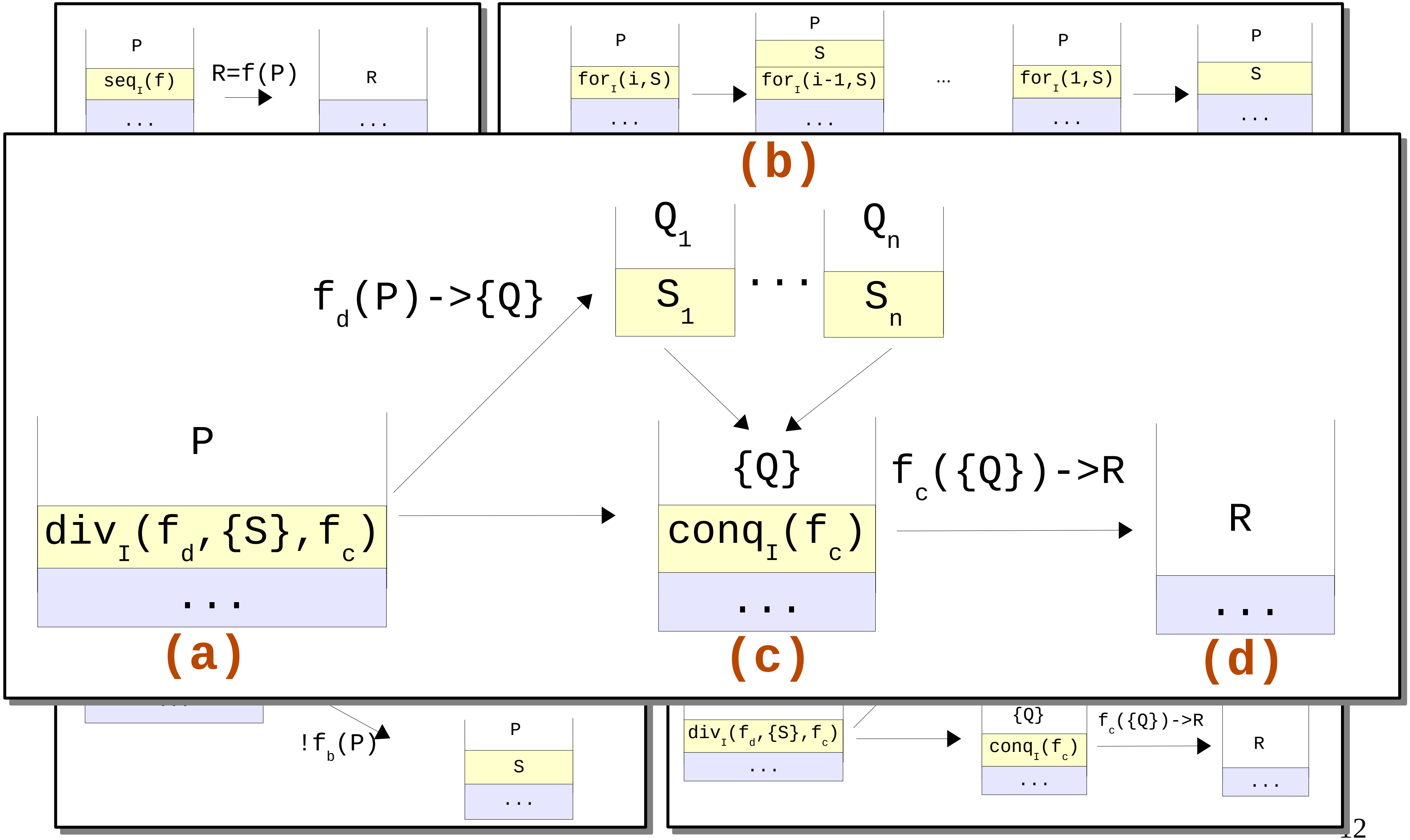
Muscles	Input		Output	Description
f_e	1	→	1	Execute
f_s	1	→	N	Split
f_m	N	→	1	Merge
f_c	1	→	boolean	Condition

Executing Skeletons

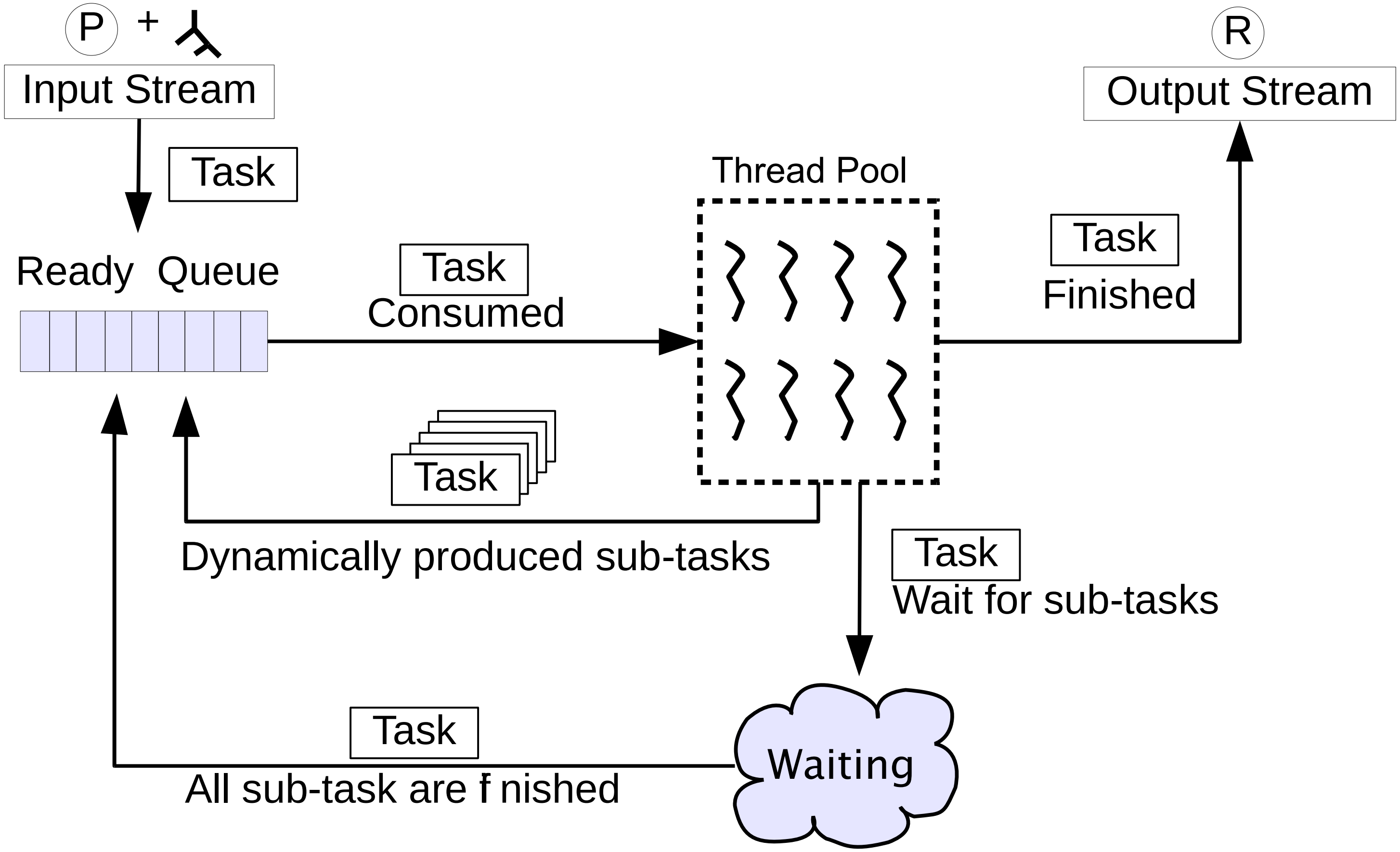
Skeletons have a lower level representation: instructions



Skeleton semantics



Skandium in a Nutshell



Agenda

- Introduction to Algorithmic Skeletons
- Our Skeletons and Libraries
- **Typing Skeletons**
- Exceptions
- Semantics for Skeletons and Exceptions
- Conclusion and Future Work

Typing Skeletons

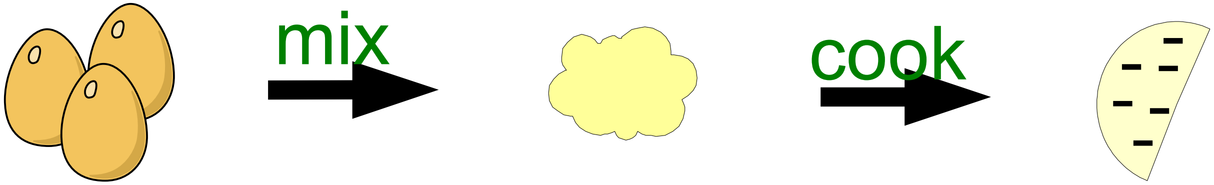
Formalisation

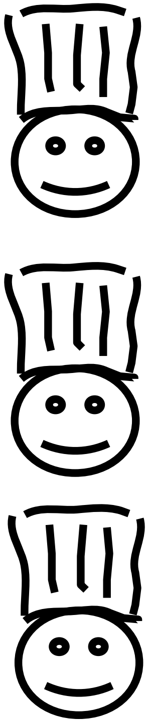
- Formal Type System for nestable Skeletons
- Proved Subject Reduction property
- Formalises type transfer between muscles

Implementation

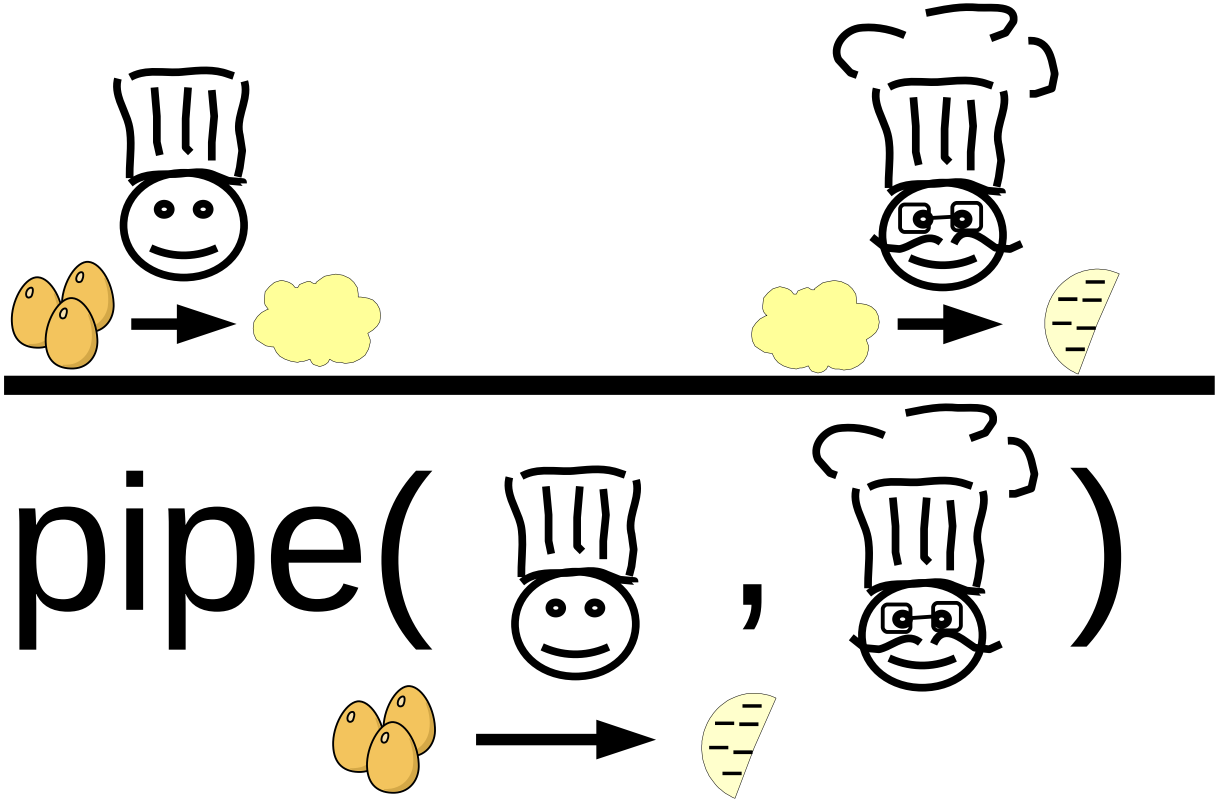
- Implemented Type System in Java
- Take advantage of Generics
- No further type casts inside muscles
- Type errors detected at skeleton composition

An Omelette Pipe



pipe( ,)

Omelette Pipe: Type Rule



pipe(,)

Do we need to check that
the input of the chef is mixed eggs?

Type System

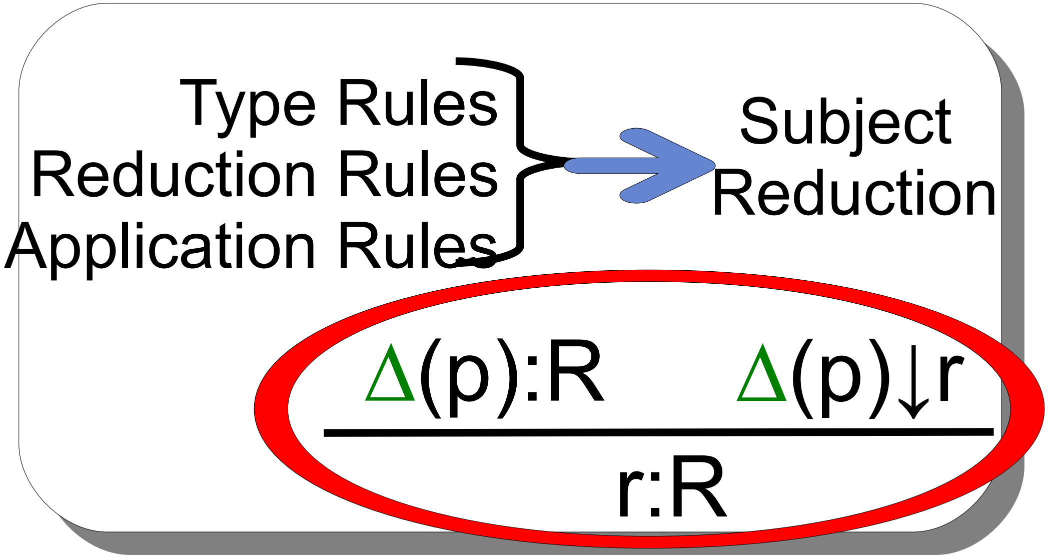
- Typing pipe

$$\frac{\Delta_1 : P \to X \qquad\qquad \Delta_2 : X \to R}{\mathrm{pipe}(\Delta_1, \Delta_2) : P \to R}$$

- Typing divide and conquer

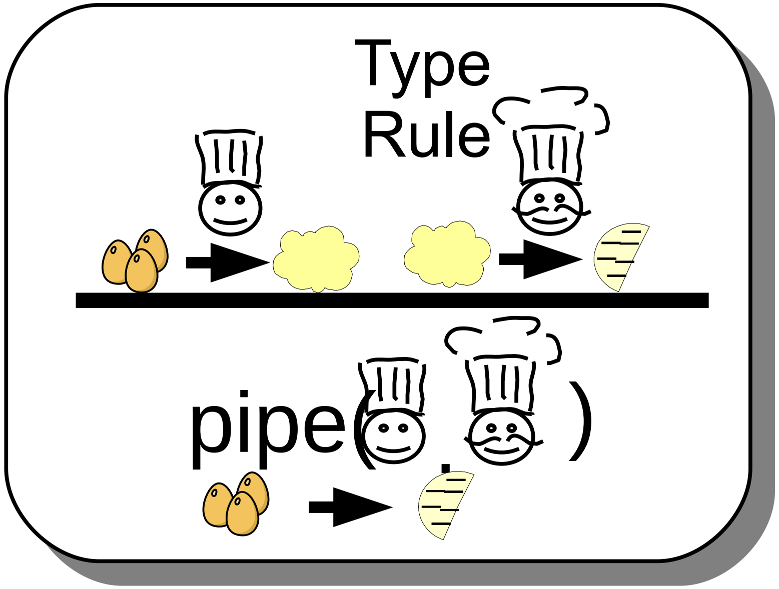
$$\frac{f_d : P \to \{P\} \quad f_b : P \to \mathrm{boolean} \quad \Delta : P \to R \quad f_c : \{R\} \to R}{\mathrm{d\&c}(f_d, f_b, \Delta, f_c) : P \to R}$$

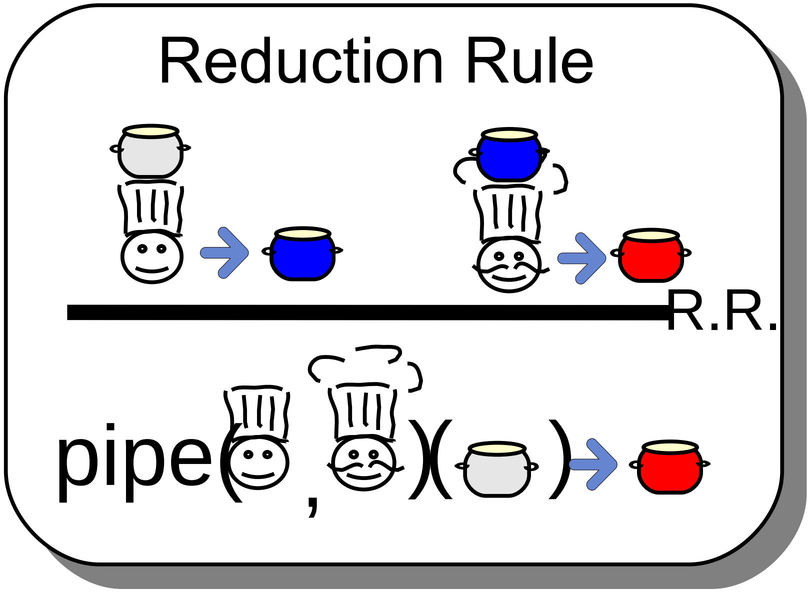
Manual check?

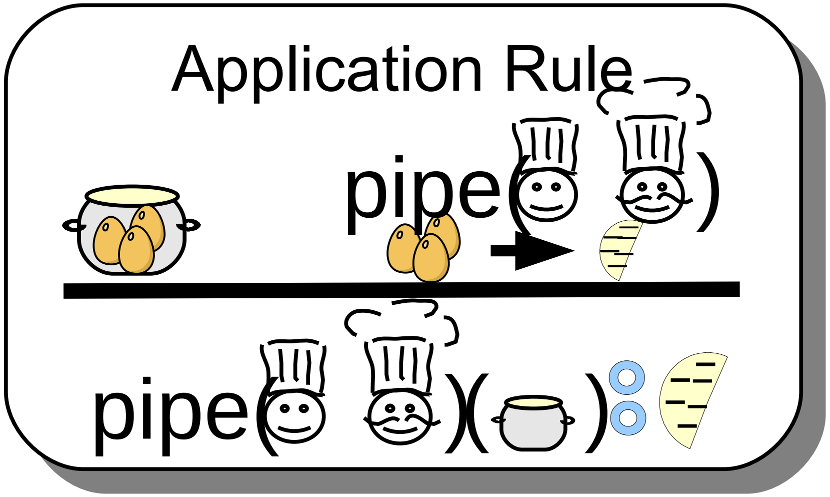


Type Rules
Reduction Rules
Application Rules
→ Subject Reduction

$$\frac{\Delta(p):R \qquad \Delta(p)\downarrow r}{r:R}$$

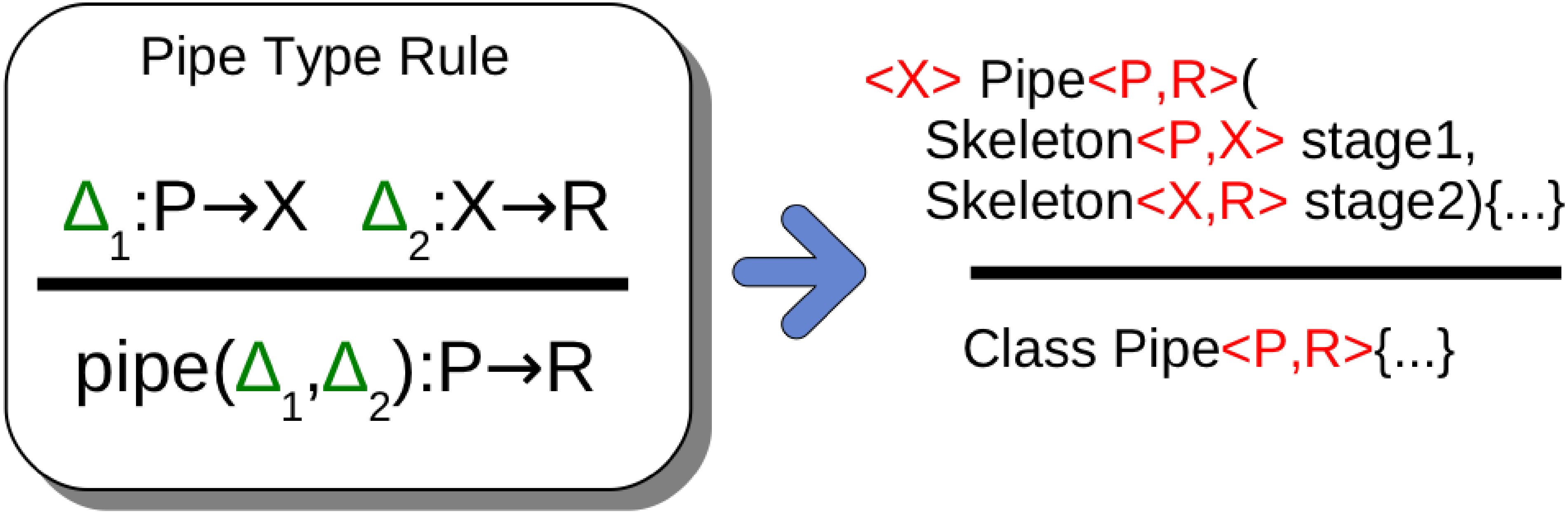






From Theory to Practice

- Use Java Generics to enforce type system
- No type validation to be imlpemented
- Same types for Java and skeletons



Pipe Type Rule

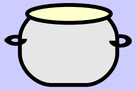
$$\frac{\Delta_1 : P \to X \quad \Delta_2 : X \to R}{pipe(\Delta_1, \Delta_2) : P \to R}$$

```
<X> Pipe<P,R>(
   Skeleton<P,X> stage1,
   Skeleton<X,R> stage2){...}
```

```
Class Pipe<P,R>{...}
```

Type Safe Muscles

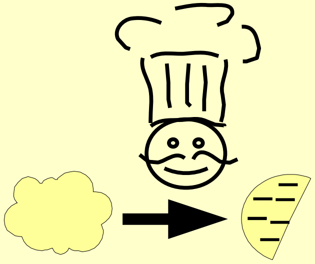
```
class Chef implements Execute{

   public Object exec(Object input){
      Mix mix = (Mix) input;
      ...
      return output;
   }
}
```



Manual check

throw new ClassCastException()

```
class Chef implements Execute<Mix, Omelette>{

   public Omelette exec(Mix input){
      ...
      return output;
   }
}
```



Agenda

- Introduction to Algorithmic Skeletons
- Our Skeletons and Libraries
- Typing Skeletons
- **Exceptions**
- Semantics for Skeletons and Exceptions
- Conclusion and Future Work

Exceptions in Skandium

- Premise: All user provided code may, at some point, generate exceptions. Such as:
  - File open, access errors
  - Code errors, such as index out of bounds
  - etc..
- Algorithmic Skeletons need to address this reality.
  - *inversion of control* + *pattern nesting* + *lower level transformations* → exceptions require a special mechanism.

BLAST Example

Skeleton Definition

```
Skeleton<Blast,File> blast =
      new DaC<Blast, File>(
            new ShouldSplit(),
            new SplitDatabase(),
            new Pipe<Blast, File>(...),
            new MergeResults()
);
```

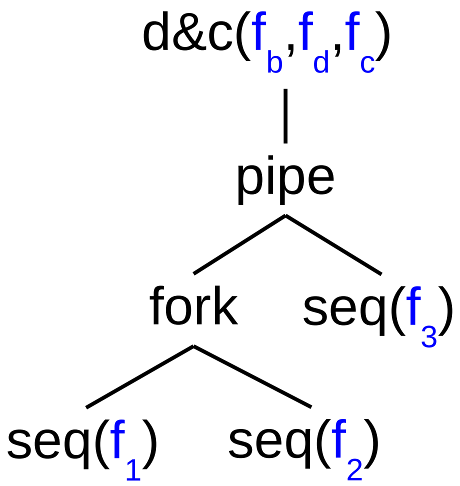
Input

```
Future<File> future =
                    blast.input(new Blast(...));
```

Output

```
File result = future.get()
```

BLAST
Skeleton Program



BLAST Example

Skeleton Definition

```
Skeleton<Blast,File> blast =
      new DaC<Blast, File>(
            new ShouldSplit(),
            new SplitDatabase(),
            new Pipe<Blast, File>(...),
            new MergeResults()
);
```

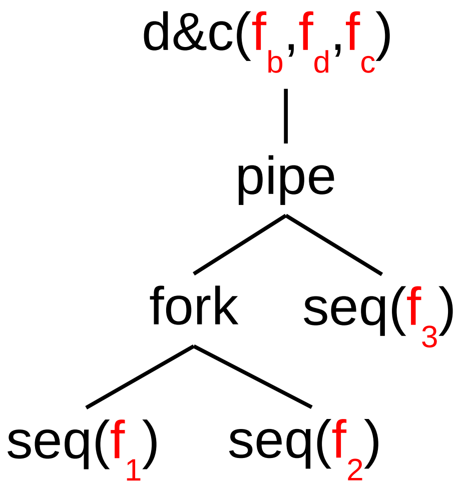
Input

```
Future<File> future =

                  blast.input(new Blast(...));
```

Output

```
File result = future.get()
```

BLAST
Skeleton Program



🟥 Exception Sources

Muscle Blast **Execute (f2)**

```
class BlastExecute implements Execute<Blast,File>{
    command = “/usr/bin/blast”;

    public File execute(Blast b) throws IOException {

       Runtime.getRuntime().exec(command + b.args);

       return new File(“result.blast”);
    }
}
```

Exception Model

Alternatives

- **Handle at the current Skeleton Level**

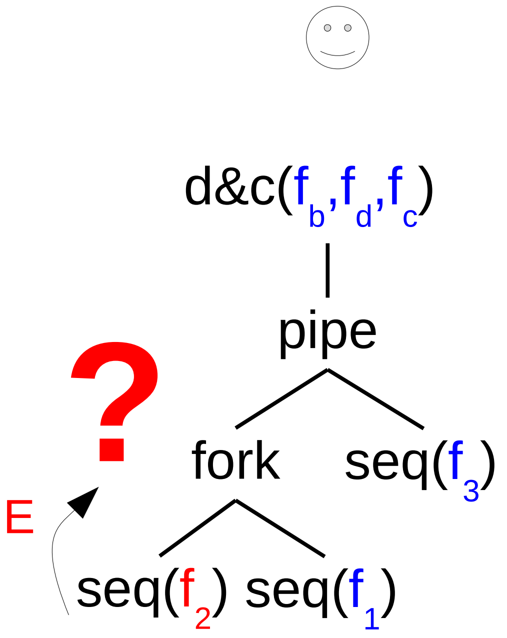
    Handler mechanism

- **Report Exception Back to Upper level**

    Handlers can be put at any level chosen by the programmer

    Return a high-level stack trace to the user if no handler

BLAST
Skeleton Program



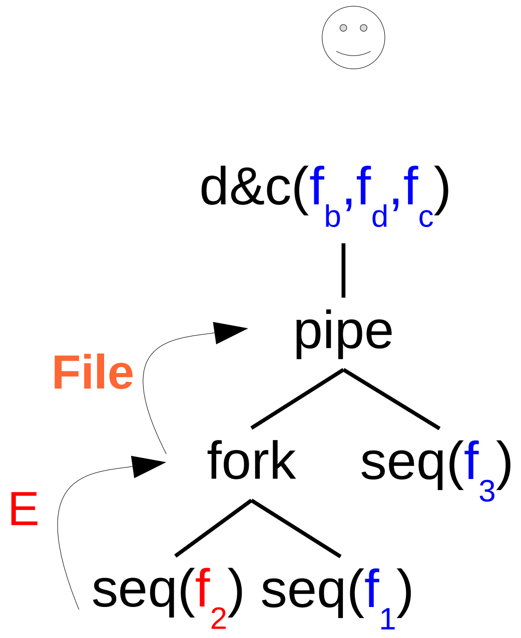
Exception Model

- **Handler**

```
fork =  new Fork(...., new IOHandler());
```

```
class IOHandler implements Handler<...>{
      command = “/usr/bin/blast”;

      public File handle(IOException e, Blast b){
         b = downloadDatabase(b);

         Runtime.(...).exec(command + b.args);

         return new File(“result.blast”);
      }
}
```

BLAST
Skeleton Program



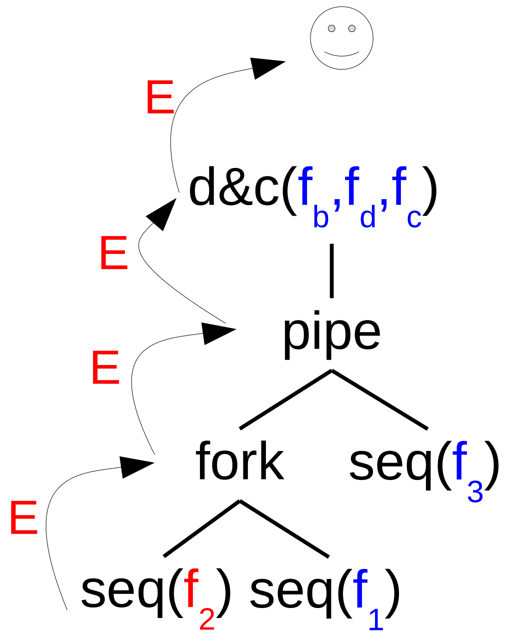
Exception Model

- **Report Exception Back to User**

Exception Stack Trace

```
Caused by: java.lang.Exception: IOException
at examples.blast.BlastExecute.execute(Blast.java:26)
at examples.blast.BlastExecute.execute(Blast.java:1)
at instructions.SeqInst.interpret(SeqInst.java:53)
at system.Interpreter.interLoop(Interpreter.java:69)
at system.Interpreter.inter(Interpreter.java:163)
at system.Task.run(Task.java:137)
```

BLAST
Skeleton Program



- Low-level information breaks high-level paradigm!
- **Nested structure information is lost!**

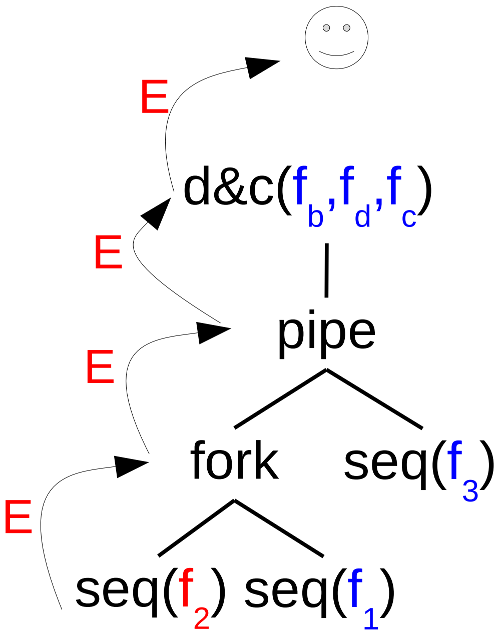
Exception Model

- **Report Exception Back to User**

BLAST
Skeleton Program

High-level Stack Trace

```
Caused by: java.lang.Exception: IOException
at examples.blast.BlastExecute.execute(Blast.java:26)
at examples.blast.BlastExecute.execute(Blast.java:1)
at skeletons.Fork.<init>(Blast.java:68)
at skeletons.Pipe.<init>(Blast.java:53)
at skeletons.DaC.<init>(Blast.java:60)
```



- Hides low-level information
- **Retraces the skeleton nesting**
- **Exceptions cancel sibling tasks**

Agenda

- Introduction to Algorithmic Skeletons
- Our Skeletons and Libraries
- Typing Skeletons
- Exceptions
- **Semantics for Skeletons and Exceptions**
- Conclusion and Future Work

A semantics for algorithmic skeletons

- To precisely and formally **define runtime behaviour**
  - **Small step** semantics for algorithmic skeletons [Leyton's PhD thesis]
  - Here: extend the semantics → define the behaviour of a skeleton with **exceptions** and **handlers**
- To formally prove properties on algorithmic skeletons
  - **Typing algorithmic skeletons** [PDP'08]:
    - Type **preservation** → skeletons can be used to transmit typed values between muscles
    - Type **errors can be detected** statically
  - … → future works

Illustrative Example

- Skeleton:

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

Translation into instructions (transform code):

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

τ_x = locations of instructions

Why an intermediate language?

- Instead of directly reducing source code
- Better reflect implementation
- Reflect simply task parallelism
- Richer language to express intermediate reduction states
  - e.g. unfold *if*:

$$if_I(f_b, S_{\text{true}}, S_{\text{false}})(p) \rightarrow seq_I(f_b)(p) \cdot choice(p, S_{\text{true}}, S_{\text{false}})$$

Notations:
Handler
● Sequence
| Parallelism

Illustrative Example

Provide input: d_1

$$pipe_I(if_I(f_b, seq_I(f_{pre}) \uparrow h_p(\tau_p), seq_I(f_{id}) \uparrow \emptyset(\tau_1)) \\ \uparrow \emptyset(\tau_i), seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau)([d_1])$$

Illustrative Example

Remember data

$$pipe_I(if_I(f_b, seq_I(f_{pre}) \uparrow h_p(\tau_p), seq_I(f_{id}) \uparrow \emptyset(\tau_1)) \\ \uparrow \emptyset(\tau_i), seq_I(f_t) \uparrow \emptyset(\tau_t))(d_1) \uparrow h(\tau, d_1))$$

Illustrative Example

Pipe reduction

$$
\begin{aligned}
&if_I(f_b, seq_I(f_{pre}) \uparrow h_p(\tau_p), seq_I(f_{id}) \uparrow \emptyset(\tau_1)) \\
&\qquad \uparrow \emptyset(\tau_i)(d_1) \odot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)
\end{aligned}
$$

Illustrative Example

Remember data+ if unfolding

$$seq_I(f_b)(d_1) \cdot choice_I(d_1, seq_I(f_{pre}) \uparrow h_p(\tau_p), seq_I(f_{id}) \uparrow \emptyset(\tau_1))$$

$$\uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1))$$

Illustrative Example

Sequential reduction:

$$true \cdot choice_I(d_1, seq_I(f_{pre}) \uparrow h_p(\tau_p), seq_I(f_{id}) \uparrow \emptyset(\tau_1))$$
$$\uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)$$

Illustrative Example

choice:

$$\boxed{seq_I(f_{pre})} \uparrow h_p(\tau_p)(d_1) \uparrow \emptyset(\tau_i, d_1) \cdot$$
$$pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)$$

Illustrative Example

Exception *e* raised:

$$e \uparrow h_p(\tau_p, d_1) \uparrow \emptyset(\tau_i, d_1) \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1)$$

Illustrative Example

Uncaught exception → trace remembered

$$\tau_i \oplus \tau_p \oplus e \cdot pipe_I(seq_I(f_t) \uparrow \emptyset(\tau_t)) \uparrow h(\tau, d_1))$$

Illustrative Example

Lookup in the next handler

$$\tau_i \oplus \tau_p \oplus e \uparrow h(\tau, d_1)$$

Illustrative Example

Exception caught

$$h(d_1, \tau_i \oplus \tau_p \oplus e)$$

Conclusion

- Theoretical (semantics+proofs) and practical (Java library)
- Typing algorithmic skeletons
  - No further type casts inside muscles
  - Type errors detected at skeleton composition
  - Skeletons transfer types between muscles
- Exceptions for Algorithmic Skeletons:
  - Featuring Support for
    - Asynchronous computations
    - Parallel Patterns Nesting
    - High level stack trace
  - Provide high-level stack trace back to users

Thank you