Advanced Behaviour

Dynamic Binders & Faults

Saverio Giallorenzo | sgiallor@cs.unibo.it

Previously on **Jolie**

```
inputPort id {
  Location: URI
  Protocol: p
  Interfaces: iface_1,
              ...,
              iface_n
}
```

```
outputPort id {
  Location: URI
  Protocol: p
  Interfaces: iface_1,
              ...,
              iface_n
}
```

OutputPorts information

Locations and **protocols** (called *binding information*) of OutputPorts can be accessed at runtime

```
outputPort MyPort {
  Location: “socket://p.com:8000"
  Protocol: sodep
  Interfaces: MyInterface
}
main
{
  println@Console(
    MyPort.location )();
// prints “socket://p.com:8000"
  println@Console(
    MyPort.protocol )()
// prints "sodep"
}
```

Dynamic Binding

Locations and **protocols** (called *binding information*) of OutputPorts can be changed at runtime

```
outputPort P {
  Interfaces: MyInterface
}

main
{
  P.location = "socket://localhost:8000";
  P.protocol = "sodep"
}
```

Dynamic Binding

The Jolie Standard Library at "`types/Binding.iol`" provides the **Biding** type

```
type Binding:void {
  .location:string
  .protocol?:string { ? }
}
```

Dynamic Binding

```
interface RegistryInterface {
   RequestResponse: getBinding( string )( Binding )
}

inputPort In {
Location: ...
Protocol: ...
Interfaces: RegistryInterface
}


main
{
   getBinding( name )( b ){
      if ( name == "LaserPrinter" ){
         b.location = "socket://p1.com:80/";
         b.protocol = "sodep"
      } else if ( name == "InkJetPrinter" ) {
         b.location = "socket://p2.it:80/";
         b.protocol = "soap"
      }
   }
}
```

Fault Handling

Four concepts behind Jolie's fault handling:

- Scopes;
- Faults;
- Throw;
- Install.

Fault Handling - Scopes

We already met **scopes** talking about parallel composition

Good practice: use **scopes** {} to explicitly group parallel statements when mixed with sequences

```
print@Console( "A" )()|
print@Console( "B" )();
print@Console( "C" )()
```

is equal to

```
{ print@Console( "A" )()|
  print@Console( "B" )()
};
print@Console( "C" )()
```

But this is easier to understand

Fault Handling - Scopes

A scope is a behavioural container denoted by a **unique name** and **able to manage faults**.

main { … } & init { … }

Are (special) scopes named main and init

Fault Handling - Scopes

A scope is a behavioural container denoted by a **unique name** and **able to manage faults**.

{ … }

This is an unnamed scope

Fault Handling - Scopes

A scope is a behavioural container denoted by a **unique name** and **able to manage faults**.

```
scope( scope_name )
{
 // code
}
```

Fault Handling - Faults

A fault is a **signal**, identified by its name, raised by a behaviour **towards the enclosing scope**.

Jolie provides the statement `throw` to raise faults.

```
scope( scope_name )
{
  // omitted code

  throw( fault_name )
}
```

Fault Handling - Faults

Jolie provides the statement `throw` to raise faults.

```
scope( division )
{
 n = 42;
 d = 0;
 if( d == 0 ) {
  throw( DivisionByZero )
 } else {
  result = n/d
 }
}
```

Will print

```
Thrown unhandled
fault: DivisionByZero
```

Fault Handling - Install

The `install` statement provides the installation of dynamic fault handlers.

```
scope( scope_name )
{
  install ( fault_name1 => /* fault handling code */,
            /* ... */  => /* fault handling code */,
            fault_nameN => /* fault handling code */
  );

  // omitted code
  throw( fault_name )
}
```

Fault Handling - Install

`install` joins a fault to a process and its handler is executed when the scope catches the fault.

```
scope( scope_name )
{
  install ( fault_name1 => /* fault handling code */,
            /* ... */  => /* fault handling code */,
            fault_nameN => /* fault handling code */
  );

  // omitted code
  throw( fault_name )
}
```

Fault Handling - Install

`install` joins a fault to a process and its handler is executed when the scope catches the fault.

```
scope( division )
{
 install( DivisionByZero =>
 println@Console(
   "Caught division by zero!" )()
 );
 n = 42;
 d = 0;
 if( d == 0 ) {
  throw( DivisionByZero )
 } else {
  result = n/d
 }
}
```

Will (gracefully) print

```
Caught division by zero!
```

Fault handling - Install priority

Jolie always **prioritises** the **install primitive** when composed in parallel with other (possibly faulty) instructions. This makes handler installation predictable.

Fault handling - Install priority

Install always goes first

```
scope( s )
{
  throw( f ) |
  install( f =>
    println@Console( "Fault caught!" )()
  )
}
```

Fault handling - Raising faults with Operations

Uncaught fault signals in a request-response body are automatically sent to the invoker. **Invokers are always notified of unhandled faults**.

RequestResponse operations declaration **can define faults (and their data type)** that could be sent back to invokers.

Fault handling - Raising faults with Operations

RequestResponse operations declaration **can define faults (and their data type)** that could be sent back to invokers.

```
interface MyInterface {
  RequestResponse:
    RR1( t1 )( t2 ) throws error1( e_type1 )
                           error2( e_type2 )
      //...

    RRN( ... )( ... ) throws errorN( e_typeN )
}
```

Fault handling - Raising faults with Operations

RequestResponse operations declaration **can define faults (and their data type)** that could be sent back to invokers.

```
type DivFaultType: void{
   .faultError: string
}

interface DivisionInterface {
 RequestResponse: divide( DivType )( int ) throws
    DivisionByZero( DivFaultType )
}
```

Fault handling - Raising faults with Operations

```
type DivType: void { .n: int .d: int } }

type DivFaultType: void { .error: string }

interface DivisionInterface {
RequestResponse: divide( DivType )( int ) throws DivisionByZero( DivFaultType )
}

inputPort In {
Location: // …
Protocol: // …
Interfaces: DivisionInterface
}

main
{
  divide( request )( request.n/request.d ){
   if( request.d == 0 ){
      throw( DivisionByZero, { .error = "You passed 0 as denominator" } )
   }
  }
}
```

Fault handling - Raising faults with Operations

```
type DivType: void { .n: int .d: int } }

type DivFaultType: void { .error: string }

interface DivisionInterface {
RequestResponse: divide( DivType )( int ) throws DivisionByZero( DivFaultType )
}

inputPort In {
Location: // …
Protocol: // …
Interfaces: DivisionInterface
}

main
{
  divide( request )( request.n/request.d ){
   if( request.d == 0 ){
      throw( DivisionByZero, { .error = "You passed 0 as denominator" } )
   }
  }
}
```

Fault handling - Raising faults with Operations

```
type DivType: void { .n: int .d: int } }

type DivFaultType: void { .error: string }

interface DivisionInterface {
RequestResponse: divide( DivType )( int ) throws
DivisionByZero( DivFaultType )
}

inputPort In {
Location: // …
Protocol: // …
Interfaces: DivisionInterface
}

main
{
  divide( request )( request.n/request.d ){
    if( request.d == 0 ){
        throw( DivisionByZero, { .error = "You
passed 0 as denominator" } )
    }
  }
}
```

Client

```
outputPort Out {
Location: // …
Protocol: // …
Interfaces: DivisionInterface
}

main
{
  req.n = 42;
  req.d = 0;
  divide@Out( req )( res )
}
```

Will print

```
Thrown unhandled
fault: DivisionByZero
```

Fault handling - Raising faults with Operations

```
type DivType: void { .n: int .d: int } }

type DivFaultType: void { .error: string }

interface DivisionInterface {
RequestResponse: divide( DivType )( int ) throws
DivisionByZero( DivFaultType )
}

inputPort In {
Location: // …
Protocol: // …
Interfaces: DivisionInterface
}

main
{
  divide( request )( request.n/request.d ){
    if( request.d == 0 ){
        throw( DivisionByZero, { .error = "You
passed 0 as denominator" } )
    }
  }
}
```

Client

```
// …
scope( divScope )
 {
  install( DivisionByZero =>
    println@Console(
     divScope.DivisionByZero
      .error )()
  );
  req.n = 42;
  req.d = 0;
  divide@Out( req )( res )
  }
}
```

Will (gracefully) print

```
You passed 0 as
denominator
```

Fault handling - Termination & Compensation

Besides using a **specific fault name**, **install** can refer to

`this` or

`default` keywords

Useful to handle **recovery**

Fault handling - Termination & Compensation

`this` refers to the enclosing scope and is used to handle its **termination**.

```
scope ( scope_name )
{
  install( this =>
    println@Console( "Recovery for scope_name" )()
  );
  println@Console( "I am scope_name" )()
}
|
throw( a_fault )
```

Fault handling - Termination & Compensation

`default` is the fallback for all faults that do not have a specific fault handler.

```
install( default =>
    println@Console( "Recovery for all faults" )()
  );
scope ( scope_name )
{
  install( this =>
    println@Console( “Recovery for scope_name" )()
  );
  println@Console( "I am scope_name" )()
}
|
throw( a_fault )
```

Fault handling - Termination & Compensation

When a scope terminates, **first** it **terminates** its own **child scopes** and **then** executers **its recovery handler**.

```
scope( granpa )
{
 install( this =>
  println@Console( “rec. granpa" )()
  );
  scope( dad )
  {
    install( this =>
     println@Console( “rec. dad" )()
    );
   scope ( son )
   {
    install( this =>
     println@Console( “rec. son" )()
    );
    sleep@Time( 500 )();
    println@Console( "son's code" )()
  }
 }
}
|
throw( a_fault )
```

Fault handling - Termination & Compensation

Recovery handlers can be **dynamically updated** like fault handlers.

```
scope( scope_name )
{
 println@Console( "step 1" )();
 install( this =>
  println@Console( "rec step 1" )() );

 println@Console( "step 2" )();
 install( this =>
  println@Console( "rec step 2" )() );

 println@Console( "step 3" )();
 install( this =>
  println@Console( "rec step 3" )() );

 println@Console( "step 4" )();
 install( this =>
  println@Console( "rec step 4" )() )
}
|
throw( a_fault )
```

Fault handling - Termination & Compensation

Recovery handlers can also be **built incrementally** via the **current handler cH**

```
scope( scope_name )
{
 println@Console( "step 1" )();
 install( this =>
  println@Console( "rec step 1" )()
 );

 println@Console( "step 2" )();
 install( this =>
  cH; println@Console( "rec step 2" )()
  );

 println@Console( "step 3" )();
 install( this =>
  cH; println@Console( "rec step 3" )()
 );

 println@Console( "step 4" )();
 install( this =>
  cH; println@Console( "rec step 4" )()
 )
}
|
throw( a_fault )
```

Fault handling - Termination & Compensation

Compensation handles the recovery of a (successfully) executed scope.

Compensation is invoked by means of the `comp` statement, which can be used only within a handler.

```
install( a_fault =>
 println@Console( “a_fault handler" )();
 comp( myScope )
);

scope( myScope )
{
 install( this =>
  println@Console( "rec step 1" )()
 );

 println@Console( "Code of myScope" )();
 install( this =>
  cH; println@Console( "rec step 2" )()
 )
};
throw( a_fault )
```

Fault handling - Termination & Compensation

Within fault handlers, Jolie provides the ^ operator to “freeze” the value of a variable.

(Useful for error reporting and debugging)

```
install( a_fault =>
 comp( example_scope )
);
scope( example_scope )
{
  install( this =>
   println@Console( "init rec" )()
  );
  i = 1;
  while( true ){
   install( this =>
    cH;
    println@Console( "rec step" + ^i)()
   );
   i++
  }
}
|
throw( a_fault )
```