Exceptions with Parameters in Poly

David C.J. Matthews

This document is a suggestion on how to put exceptions with parameters into Poly. The idea is essentially that suggested by Alan Mycroft for Standard ML.

1 Background

This can be skipped by those familiar with the ML exception mechanism.

If we want exceptions to have parameters we need to guarantee when we raise an exception with some parameters, that when we catch it, the parameters have the signatures we expect. A simple way of doing this is to introduce an exception declaration such as `exception x(string);`. This exception can be raised with a string parameter `raise x("hello");` and caught with a handler expecting a string `catch x(p:string).print p;`. The type checker can easily check that in both cases a string is being used.

The problem is to guarantee that the `x` we catch is the same as the `x` we raised. We need some identification when we raise an exception which the handler can look at to decide which exception this is. If we can declare local exceptions with scope in a similar way to local identifiers we may have two exceptions both called `x` with different argument signatures in separate parts of the same program. We could pass a procedure which raises one `x` but does not catch it into the scope of a handler for the other `x`. Since the parameters have different signatures the handler must treat this as a different exception even though it has the same name. This problem can be avoided if the compiler generates a unique identification for each exception. Unfortunately this will not work properly either, if we have exceptions with polymorphic parameters.

```
let p ==
proc(t: type end; b: boolean; x:t; f: proc(t)t)t
   begin
   let ex == exception(t); { Declare exception x }
```

```
   if b then raise ex(x)   { Raise the exception }
   else
      begin
      f(x) { Call f, which may raise an exception }
      catch ex(y: t). y { Catch ex, returning the parameter. }
      end
   end;
let q ==
proc(i: integer)integer
   begin
   let z ==
      p(string,
        true, { Raise an exception with a string parameter. }
        "hello", { A string }
        proc(s: string)string (s) { Not used }
       );
   i+1
   end;
let result: integer ==
   p(integer,
     false, { Catch an exception }
     1,     { Unused }
     q      { proc(integer)integer but raising ex with a string }
     );
```

p is a procedure which either raises an exception or calls a procedure depending on the value of b. It is polymorphic since it is parameterised on the type, t, used in the signatures of x and f and in the result signature. Within q t is string and so the exception ex is raised with a string argument. In the second call to p the type is integer. This time p is catching the exception, but expects an integer and returns it as an integer. If the exception is the same in both cases so that the exception raised in one call to p can be caught in a different call the type system can be broken. The only way to keep the type system secure is to say that different invocations of p generate different exceptions. This requires the exception declaration to create a value dynamically which will be the exception identification for ex in that particular scope. Within the scope all references to ex will find the same identification, but if the exception declaration is executed again a different identification will be created, and the exceptions will not match.

2 Proposal

We introduce a new class of signature, the *exception* to rank alongside types, procedures and simple values. An exception has an argument list exactly like a procedure, but no result signature. The argument list can include implied arguments. A missing argument list is treated as though a single argument of type *void* had been given.

Syntax		
<signature>	::=	**exception** <argument list> |
		exception

```
rangeerror: exception
failure: exception(string)
print_ex: exception[t: type (t) print: proc(t) end](x:t)
```

An exception is a value and is created by the *exception constructor.*

Syntax		
<expression>	::=	**exception** <argument list> |
		exception

```
let rangeerror == exception
let failure == exception(string)
let print_ex == exception[t: type (t) print: proc(t) end](x:t)
```

It will normally only be used in a declaration but it is possible to pass it as a parameter or to return it from an if-statement or a procedure. It can also be a component of a type.

An exception is raised with the *raise expression.* The arguments supplied in the expression must match the signature of the exception. The rules are the same as for a procedure.

Syntax		
<expression>	::=	**raise** <identifier> <expression> |
		raise <identifier>

```
raise rangeerror
raise failure "It failed"
raise print_ex 2
```

It would be possible to have an arbitrary expression in place of the identifier but it seems so unlikely that, for the moment at any rate we will keep this restriction. The only case that is at all likely is a selection from a type. Having this restriction also allows us to have a name to print if we catch an exception in an **else** catch phrase.

An exception is caught with the *catch phrase* at the end of a compound expression.

Syntax		
<handler>	::=	**catch** <catch phrase>; ... ; <catch phrase>
<catch phrase>	::=	<identifier> <parameter list> . <expression> |
		<identifier> . <expression> |
		else <parameter list> . <expression> |
		else . <expression>

```
catch rangeerror          . print "Overflow";
      failure (x: string) . print("Failed with " + x);
      print_ex [t: type (t) print: proc(t) end](x:t)
  . (print "Exception "; print x);
      else (x: string)   . (print "Unknown "; print x)
```

The general form of the catch phrase consists of a named exception followed by a parameter list and an expression to execute. As with the raise expression this could be generalised to allow arbitrary exception-returning expressions. When an exception is raised the exception value, *not* the actual name, is compared with the values in each of the catch phrases until a match is found. The actual parameters given in the raise expression are used as the values in the parameter list, if there is one, and the corresponding expression is evaluated. The signature of the parameter list must match the signature of the exception.

The **else** form of the catch phrase catches all exceptions. The parameter list if present must have a single string parameter. If it catches an exception that parameter will be set to the name of the exception used in the *raise expression.*