CS520, Handout 13, 17 October 2006

(Updated 19 October 2006)

Exceptions

Reading: [Pierce, Chapter 14]

Examples of Exceptions

In ML

Declaration of an exception:

```
- exception Factorial;     (* has type 'exn' *)
exception Factorial
```

A function that raises but does not handle an exception:

```
- fun checked_factorial n =
    if n < 0 then raise Factorial else
      if n=0 then 1 else n * checked_factorial (n-1);
val checked_factorial = fn : int -> int

- checked_factorial 3;
val it = 6 : int

- checked_factorial ~3;
uncaught exception
```

In ML

Handling an exception:

```
- checked_factorial ~3 handle Factorial => 0;
val it = 0 : int;
```

A function that raises and handle an exception:

```
- fun checked_factorial2 n =
    (if n < 0 then raise Factorial else
        if n=0 then 1 else n * checked_factorial2 (n-1))
    handle Factorial  =>  checked_factorial2 (0 - n);
val checked_factorial2 = fn : int -> int

- checked_factorial2 3;
val it = 6 : int

- checked_factorial2 ~3;
val it = 6 : int
```


In ML

The type `exn` provided by ML includes the following pre-defined constructors:

```
datatype exn =
      ...
    | Division_by_zero
    | Failure of string
    | ...
```

In contrast to ordinary types, the type `exn` can be extended. It is the only case of a ***extensible*** **variant type** in ML. We can add user-defined constructors, by declaring:

```
- exception Failed;
exception Failed

- exception FailedBecause of string;
exception FailedBecause of string

- exception BadValue of int;
exception BadValue of int

- exception BadMath of string * int * int;
exception BadMath of string * int * int
```

The result of the preceding declarations is to extend the type `exn` as follows:

```
datatype exn =
      ...
    | Division_by_zero
    | Failure of string
    | ...
    | Failed
    | FailedBecause of string
    | BadValue of int
    | BadMath of string * int * int
```


In the language of [Pierce], the preceding type declaration can be de-sugared as the following “extensible” variant type:

```
Exn = < ... ,
        Division_by_zero : Unit,
        Failure : String,
        ... ,
        Failed : Unit,
        FailedBecause : String,
        BadValue : Int,
        BadMath : String * Int * Int
      >
```

where the last 4 fields are user-defined. I capitalize the first letter in `Exn` to follow Pierce’s convention for naming types. The definition of `Exn` above also assumes that the base types `String` and `Int` have been added to the language of [Pierce].

In ML

An example of a function that raises the user-defined `BadMath` exception:

```
fun avg (sum,count) =
   if count = 0 then raise BadMath("div", sum, count)
                else sum div count;

fun saveAverage (sum,count) =
   avg (sum,count)
   handle BadMath (s,a,b) => (
     print("Math error: attempted "); print(s); print(" of");
     print(Int.toString(a)); print(" and "); print(Int.toString(b));
     print("Result of zero used.");
     print("\n");
     0           (* return value! *)
   );
```


In OCaml

```
# exception Failure of string;;
exception Failure of string

# Failure "oops";;
- : exn = Failure "oops"

# let hd = function
     [] -> raise (Failure "hd")
     | (a :: l) -> a;;
val hd : 'a list -> 'a = <fun>

# (* raise an exception without handling it *)
  hd [];;
Exception: Failure "hd".

# (* raise and handle an exception *)
  try hd [] with Failure "hd" -> 0;;
- : int = 0
```

In JAVA

(from Arnold, Gosling and Holmes, *The Java Programming Language*, page 203)

```
class SuperException extends Exception {}
class SubException extends SuperException {}

class BadCatch {
  public void goodTry() {
    try {
      throw new SubException();
    } catch (SuperException superRef) {
      // Catches both SuperException and SubException
    } catch (SubException subRef) {
      // This would never be reached
    }
  }
}
```


In JAVA

(from Arnold, Gosling and Holmes, *The Java Programming Language*, page 203)

```
class SuperException extends Exception {}
class SubException extends SuperException {}

class BadCatch {
  public void goodTry() {
    try {
      throw new SubException();
    } catch (SuperException superRef) {
      // Catches both SuperException and SubException
    } catch (SubException subRef) {
      // This would never be reached
    }
  }
}
```

Remark: The compiler will in fact not accept the preceding code, because you cannot put a superclass `catch` clause before a `catch` of one of its subclasses. The `catch` clauses are examined in order, so a `catch` that picked up one exception type before a `catch` for an extended type of exception would be a mistake.

Exceptions that Abort Evaluation [Pierce, Sect 14.1]

Syntax of Terms

t	$::=$	$\ldots$	*terms*
		`error`	*run-time error*

Evaluation Rules

$\ldots$

$$\texttt{error}\ t \Rightarrow \texttt{error} \qquad \text{(E-AppErr1)}$$

$$v\ \texttt{error} \Rightarrow \texttt{error} \qquad \text{(E-AppErr2)}$$

Typing Rules

$\ldots$

$$\frac{}{\Gamma \vdash \texttt{error} : T} \qquad \text{(T-Error)}$$

Exceptions that Do Not Abort Evaluation [Pierce, Sect 14.2]

Syntax of Terms

t	$::=$	$\ldots$	*terms*
		`error`	*run-time error*
		`try` t `with` t	*trap error*

Evaluation Rules

$\ldots$

$$\texttt{error}\ t \Rightarrow \texttt{error} \qquad \text{(E-AppErr1)}$$

$$v\ \texttt{error} \Rightarrow \texttt{error} \qquad \text{(E-AppErr2)}$$

$$\texttt{try}\ v\ \texttt{with}\ t \Rightarrow v \qquad \text{(E-TryV)}$$

$$\texttt{try error with}\ t \Rightarrow t \qquad \text{(E-TryError)}$$

$$\frac{t_1 \Rightarrow t_1'}{\texttt{try}\ t_1\ \texttt{with}\ t_2 \Rightarrow \texttt{try}\ t_1'\ \texttt{with}\ t_2} \qquad \text{(E-Try)}$$

Typing Rules

$\ldots$

$$\frac{}{\Gamma \vdash \texttt{error} : T} \qquad \text{(T-ERROR)}$$

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \texttt{try}\ t_1\ \texttt{with}\ t_2 : T} \qquad \text{(T-TRY)}$$

Exceptions that Carry Values [Pierce, Sect 14.3]

Syntax of Terms

t ::=	$\ldots$	*terms*
	$\texttt{raise}\ t$	*raise exception*
	$\texttt{try}\ t\ \texttt{with}\ t$	*handle exception*

Evaluation Rules

$\ldots$

$(\texttt{raise}\, v)\ t \Rightarrow \texttt{raise}\, v$ (E-AppRaise1)

$v_1\ (\texttt{raise}\, v_2) \Rightarrow \texttt{raise}\, v_2$ (E-AppRaise2)

$\dfrac{t \Rightarrow t'}{\texttt{raise}\, t \Rightarrow \texttt{raise}\, t'}$ (E-Raise)

$\texttt{raise}\,(\texttt{raise}\, v) \Rightarrow \texttt{raise}\, v$ (E-RaiseRaise)

$\texttt{try}\ v\ \texttt{with}\ t \Rightarrow v$ (E-TryV)

$\texttt{try}\ (\texttt{raise}\, v)\ \texttt{with}\ t \Rightarrow t\ v$ (E-TryRaise)

$\dfrac{t_1 \Rightarrow t_1'}{\texttt{try}\ t_1\ \texttt{with}\ t_2 \Rightarrow \texttt{try}\ t_1'\ \texttt{with}\ t_2}$ (E-Try)

Typing Rules

$\ldots$

$\dfrac{\Gamma \vdash t : \texttt{Exn}}{\Gamma \vdash \texttt{raise}\, t : T}$ (T-Raise)

$\dfrac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : \texttt{Exn} \to T}{\Gamma \vdash \texttt{try}\ t_1\ \texttt{with}\ t_2 : T}$ (T-Try)

where you can take $\texttt{Exn}$ as a distinguished base type on a first reading, and then, more precisely, as the variant type defined on page 6 of this handout. For further details on $\texttt{Exn}$, see [Pierce, page 177, point 4].

A different formulation of the typing rule for `try-with` is the following (which is used in the 3rd implementation assignment):

$$\frac{\Gamma \vdash t_0 : T \quad \Gamma \vdash \mathtt{pat}_1 : \mathtt{Exn} \quad \Gamma \vdash t_1 : T \quad \cdots \quad \Gamma \vdash \mathtt{pat}_n : \mathtt{Exn} \quad \Gamma \vdash t_n : T}{\Gamma \vdash (\mathtt{try}\ t_0\ \mathtt{with}\ \mathtt{pat}_1 \Rightarrow t_1 \mid \cdots \mid \mathtt{pat}_n \Rightarrow t_n) : T} \quad \text{(Ty-Try)}$$

Exercise: Show that if we take `Exn` to be the following variant type:

```
Exn = <pat_1 : Unit, ... , pat_n : Unit>
```

then rule (Ty-Try) above is a special case of the rule (T-Try) in the preceding page.

Progress + Preservation

For the language of page 10 in this handout. Stated but not proved in [Pierce, Chapt 14, page 173].

Progress
If t is a closed well-typed term, i.e., $\varnothing \vdash t : T$ for some type T, then either t is a value or $t = \mathtt{error}$ or $t \Rightarrow t'$ for some term t'.

Preservation
If $\Gamma \vdash t : T$ and $t \Rightarrow t'$, then $\Gamma \vdash t' : T$.

Progress + Preservation

For the language of page 10 in this handout. Stated but not proved in [Pierce, Chapt 14, page 173].

Progress

If t is a closed well-typed term, i.e., $\varnothing \vdash t : T$ for some type T, then either t is a value or $t = \texttt{error}$ or $t \Rightarrow t'$ for some term t'.

Preservation

If $\Gamma \vdash t : T$ and $t \Rightarrow t'$, then $\Gamma \vdash t' : T$.

Exercise:

1. State the Progress and Preservation theorems for the language of pages 11-12 in this handout (also [Pierce, Sect 14.2]).
2. State the Progress and Preservation theorems for the language of pages 13-14-15 in this handout (also [Pierce, Sect 14.3]).