Side-Effects: Exceptions

February 21, 2001

CS 131: Programming Languages

Two Syntactic Classes

- Many languages distinguish between *expressions* and *statements/commands*.
  - Expressions are evaluated to compute a value
  - Statements are executed to change machine state
    - E.g., assignments or I/O or control flow
- Sometimes an analogous distinction between *functions* and *procedures/subroutines*
  - Whether a value is returned, or not.
  - That is, whether a call is an expression or a statement.

Side-effects

- If an expression does anything other than return a value, it is said to have a *side-effect*.
  - Assignment, I/O, raising exceptions, ...
- Some people argue effects should be avoided
  - Either just in expressions, or entirely
  - Why?
    - Easier to reason about program changes
    - More scope for compiler to optimize/parallelize
    - Smart compiler can do just as well

Arguments for Side-Effects

- Sometimes simply more convenient
- Don't have to depend upon smart compiler to recognize simulated side-effects
- Example: Haskell compiler
  - Type inference speedup
  - Inliner convolutions

Side-Effects in SML

- Sequencing
- `print`
- Exceptions
- Continuations
- References (assignment)

Sequencing in SML

- As part of an *expression*, semicolon acts like the comma operator in C.
  - The expression ($expr_1$; $expr_2$) evaluates $expr_1$, then throws away the result and evaluates $expr_2$.

Printing

- Canonical side-effecting function

```
print : string -> unit
```

- Can tell just by looking at its type that it probably has a side-effect
  - Returns no useful value

```
fun f () = (print "hello "; print "world\n")
```

Exceptions Summarized

- Way to gracefully abort a computation
- Languages supporting exceptions normally have
  - Way to create exceptions
  - Way to raise/throw an exception
  - Way to handle/catch exceptions

Exceptions in SML

- Way to create exceptions
  - An exception in SML is a value of type `exn`
  - This type is sometimes an called an *extensible datatype*
    - Has constructors like an SML datatype
    - But unlike a normal datatype, we can add new cases whenever we want
    - New exceptions declared with `exception`

Exceptions in SML

- Way to create exceptions
  - For example, after

```
exception Oops
exception Ouch of string
```

  we have

```
Oops                   :  exn
(Ouch "Slipped disk")  :  exn
```

Exceptions in SML

- Way to raise/throw an exception
  - In SML, the keyword is `raise`
  - For example,

```
raise Oops
```

  or

```
raise (Ouch "Something broke")
```

  - What should the type of `raise` be?

`raise : exn -> ???`

Exceptions in SML

- Way to handle/catch an exception
  - SML uses the `handle` keyword

<*expr*> `handle` <*pattern*$_1$> => <*handler*$_1$>
| ...
| <*pattern*$_n$> => <*handler*$_n$>

  - Meaning:
    - Evaluate <*expr*>. If it returns a value ignore the handlers and return this value.
    - Otherwise, evaluate the first handler matching the exception that was raised
    - If no handler matches, the exception keeps going.

Examples

```
print (Int.toString (compute 0))
   handle Div => print "Divide by zero"

print (Int.toString (compute 0))
   handle Div => print "Divide by zero"
        | Overflow => print "Overflow"

print (Int.toString (compute 0))
   handle Div => print "Divide by zero"
        | Ouch s => print s
```

Examples

```
print (Int.toString (compute 0))
   handle _ => ()

print (Int.toString (compute 0))
   handle Div => print "Divide by zero"
        | _ => print "Caught exception"

print (Int.toString (compute 0))
   handle Div => print "Divide by zero"
        | e => (print "Saw an exception";
                raise e)
```

A Fancy Example

- Choosing coins with a given sum
  - For example, assume you have 5-cent and 2-cent coins; how to make 8 cents?
- Problem: define the function

```
coins : int list * int -> int list
```

so that, for example,

```
coins ([5,2], 8)
```

yields

`[2,2,2,2]`.

A Fancy Example

- A greedy algorithm

```
exception Impossible
fun coins (_,0)     = []
  | coins ([],_)    = raise Impossible
  | coins (c::cs,n) =
      if (c <= n) then
         c :: (coins(c::cs,n-c))
      else
         coins(cs,n)
```

A Fancy Example

- A greedy algorithm

```
exception Impossible
fun coins (_,0)     = []
  | coins ([],_)    = raise Impossible
  | coins (c::cs,n) =
      if (c <= n) then
         (c :: (coins(c::cs,n-c)))
      else
         coins(cs,n)
```

- Problem: this doesn't work for the input ([5,2],8).

A Fancy Example

- A *backtracking* algorithm

```
exception Impossible
fun coins (_,0)     = []
  | coins ([],_)    = raise Impossible
  | coins (c::cs,n) =
      if (c <= n) then
         ((c :: (coins(c::cs,n-c))
            handle Impossible => coins(cs,n))
      else
         coins(cs,n)
```

Formal Semantics for Exceptions

- We consider the case where there is exactly one exception in the language.

e ::=   ...
      | fail
      | catch e_1 with e_2

Formal Semantics for Exceptions

- We consider the case where there is exactly one exception in the language.

e ::=   ...
      | fail
      | catch e_1 with e_2

Raise the exception

Return value of $\mathbf{e_1}$ unless it fails, in which case evaluate $\mathbf{e_2}$

Static Semantics

$$\frac{}{\Gamma \vdash \texttt{fail} : t}$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \texttt{catch } e_1 \texttt{ with } e_2 : t}$$

Dynamic Semantics

$$\frac{e_1 \to e_1'}{\texttt{catch } e_1 \texttt{ with } e_2 \to \texttt{catch } e_1' \texttt{ with } e_2}$$

$$\frac{}{\texttt{catch } v \texttt{ with } e_2 \to v}$$

$$\frac{}{\texttt{catch fail with } e_2 \to e_2}$$

$$\frac{}{\texttt{fail + } e_2 \to \texttt{fail}} \qquad \frac{}{v_1 \texttt{ + fail} \to \texttt{fail}}$$

$$\frac{}{\texttt{if fail then } e_2 \texttt{ else } e_3 \to \texttt{fail}}$$

etc.

Handling Division

$$\frac{e_1 \to e_1'}{e_1 \texttt{ div } e_2 \to e_1' \texttt{ div } e_2}$$

$$\frac{e_2 \to e_2'}{v_1 \texttt{ div } e_2 \to v_1 \texttt{ div } e_2'}$$

$$\frac{n_2 \texttt{ <> } 0}{n_1 \texttt{ div } n_2 \to n_1 \div n_2}$$

$$\frac{}{n_1 \texttt{ div 0} \to \texttt{fail}}$$

Example Evaluation 1

- `catch (3 + (2 + (6 div (3 - 3))))`
  `with (4 + 7)`
- `catch (3 + (2 + (6 div 0))) with (4 + 7)`
- `catch (3 + (2 + fail)) with (4 + 7)`
- `catch (3 + fail) with (4 + 7)`
- `catch fail with (4 + 7)`
- `(4 + 7)`
- `11`

Example Evaluation 2

- `catch (3 + (2 + (6 div (3 - 1))))`
  `with (4 + 7)`
- `catch (3 + (2 + (6 div 2))) with (4 + 7)`
- `catch (3 + (2 + 3)) with (4 + 7)`
- `catch (3 + 5) with (4 + 7)`
- `catch 8 with (4 + 7)`
- `8`

Proving Type Soundness

- Which, if any, are no longer true?
  - Inversion
    if $\Gamma \vdash$ `e`$_1$ `+ e`$_2$ `: t` then `t = Int` and $\Gamma \vdash$ `e`$_1$ `: Int` and $\Gamma \vdash$ `e`$_2$ `: Int`
  - Type Preservation
    if $\vdash$ `e : t` and `e` $\to$ `e'` then $\vdash$ `e' : t`
  - Canonical Forms
    if $\vdash$ `v : Int` then `v` is an integer constant.
  - Progress
    if $\vdash$ `e : t` then either `e` is a value or else
    `e` $\to$ `e' for some e' : t`

Proving Type Soundness

- The following variant of Progress can be proved:

  If $\vdash$ `e : t` then either
  `e` is a value
  or else `e` $\to$ `e'` for some `e'`
  or else `e = fail`.

- Hence if $\vdash$ `e : t`, one of the following is true

`e` $\to^*$ `v`	`(normal termination)`
`e` $\to^*$ `fail`	`(uncaught exception)`
`e` $\to$ `e'` $\to$ `e''` $\to$ `e'''` $\to \ldots$	`(nontermination)`

Scoping of Handlers

- In some texts you will find exception handlers as being dynamically scoped.
  - Why?
  - What would statically scoped handlers look like?

Problem

- Given SML expressions `e1 : t` and `e2 : unit`, give a piece of SML code (abbreviated `try e1 finally e2`) satisfying the following:
  - `try e1 finally e2` Should have type *t*, the same type as `e1`.
  - This code should evaluate `e1` and then regardless of exceptions evaluate `e2`.
  - If `e2` raises an exception then the whole code should raise that exception; otherwise, after `e2` finishes the code should either return the result of `e1` or raise the exception raised by `e1`, as appropriate.

- Extra credit:
  - The expressions `e1` and `e2` could be large, and it is generally bad style to duplicate large pieces of source code; find a solution that mentions `e1` and `e2` exactly once each.