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 `(expr₁ ; expr₂)` evaluates `expr₁`, then throws away the result and evaluates `expr₂`.

Printing

- Canonical side-effecting function

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

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

  ```ml
  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 \texttt{exn}
  – This type is sometimes an called an \textit{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 \texttt{exception}

• Way to raise/throw an exception
  – In SML, the keyword is \texttt{raise}
  – For example, after
    
    \begin{verbatim}
    exception Oops
    exception Ouch of string
    \end{verbatim}

    we have
    
    \begin{verbatim}
    Oops : exn
    (Ouch "Slipped disk") : exn
    \end{verbatim}

  – What should the type of \texttt{raise} be?
    
    \texttt{raise : exn -> ???}

• Way to handle/catch an exception
  – SML uses the \texttt{handle} keyword
    
    \begin{verbatim}
    <expr> handle <pattern> => <handler>
    | ...                              
    | <pattern> => <handler>
    \end{verbatim}

    – Meaning:
      • Evaluate \texttt{<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

```haskell
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
```

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,
  ```hs
  coins ([5,2], 8)
  yields [2,2,2,2].
  ```

Examples

```haskell
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

- A greedy algorithm

```haskell
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

```verbatim
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

```verbatim
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.

```verbatim
e ::= ...
| fail
| catch e₁ with e₂
```

Formal Semantics for Exceptions

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

```verbatim
e ::= ...
| fail
| catch e₁ with e₂
```

• Raise the exception

• Return value of \( e₁ \) unless it fails, in which case evaluate \( e₂ \)
Static Semantics

\[ \Gamma \vdash \text{fail} : \tau \]
\[ \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \]
\[ \Gamma \vdash \text{catch } e_1 \text{ with } e_2 : \tau \]

Dynamic Semantics

\[ e_1 \rightarrow e_1' \]
\[ \text{catch } e_1 \text{ with } e_2 \rightarrow \text{catch } e_1' \text{ with } e_2 \]
\[ \text{catch } v \text{ with } e_2 \rightarrow v \]
\[ \text{catch } \text{fail} \text{ with } e_2 \rightarrow e_2 \]
\[ \text{fail} + e_2 \rightarrow \text{fail} \]
\[ v_1 + \text{fail} \rightarrow \text{fail} \]
\[ \text{if } \text{fail} \text{ then } e_2 \text{ else } e_3 \rightarrow \text{fail} \]  

Handling Division

\[ e_1 \rightarrow e_1' \]
\[ e_1 \text{ div } e_2 \rightarrow e_1' \text{ div } e_2 \]
\[ e_2 \rightarrow e_2' \]
\[ v_1 \text{ div } e_2 \rightarrow v_1 \text{ div } e_2' \]
\[ n_1 \leftrightarrow 0 \]
\[ n_1 \text{ div } n_2 \rightarrow n_1 \odot n_2 \]
\[ n_1 \text{ div } 0 \rightarrow \text{fail} \]

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
    \[ \text{if } \Gamma \vdash e_1 + e_2 : t \text{ then } t = \text{Int} \text{ and } \Gamma \vdash e_1 : \text{Int} \text{ and } \Gamma \vdash e_2 : \text{Int} \]
  - Type Preservation
    \[ \text{if } \Gamma \vdash e : t \text{ and } e \rightarrow e' \text{ then } \Gamma \vdash e' : t \]
  - Canonical Forms
    \[ \text{if } \Gamma \vdash v : \text{Int} \text{ then } v \text{ is an integer constant.} \]
  - Progress
    \[ \text{if } \Gamma \vdash e : t \text{ then either } e \text{ is a value or else } e \rightarrow e' \text{ for some } e' : t \]

Proving Type Soundness

- The following variant of Progress can be proved:

  \[ \text{if } \Gamma \vdash e : t \text{ then either} \\
  \text{e is a value} \\
  \text{or else } e \rightarrow e' \text{ for some } e' \\
  \text{or else } e = \text{fail}. \]

- Hence if \( \Gamma \vdash e : t \), one of the following is true
  \[ e \rightarrow^* v \quad \text{(normal termination)} \]
  \[ e \rightarrow^* \text{fail} \quad \text{(uncaught exception)} \]
  \[ e \rightarrow e' \rightarrow e'' \rightarrow e''' \rightarrow \ldots \quad \text{(nontermination)} \]

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 \( e_1 : t \) and \( e_2 : \text{unit} \), give a piece of SML code (abbreviated \( \text{try } e_1 \text{ finally } e_2 \) ) satisfying the following:
  - \( \text{try } e_1 \text{ finally } e_2 \) Should have type \( t \), the same type as \( e_1 \).
  - This code should evaluate \( e_1 \) and then regardless of exceptions evaluate \( e_2 \).
  - If \( e_2 \) raises an exception then the whole code should raise that exception; otherwise, after \( e_2 \) finishes the code should either return the result of \( e_1 \) or raise the exception raised by \( e_1 \), as appropriate.

• Extra credit:
  - The expressions \( e_1 \) and \( e_2 \) could be large, and it is generally bad style to duplicate large pieces of source code; find a solution that mentions \( e_1 \) and \( e_2 \) exactly once each.