Computer Science 131
Programming Languages

September 26, 2000
Side-Effects: Exceptions
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
- References and assignment
Sequencing in SML

• As part of an expression, semicolon acts like the comma operator in C.
  - The expression \((\text{expr}_1 \; ; \; \text{expr}_2)\)
    evaluates \(\text{expr}_1\), then throws away the result and evaluates \(\text{expr}_2\).
Printing

• Canonical side-effecting function

\[
\text{print} : \text{string} \rightarrow \text{unit}
\]

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

Example:

\[
\text{fun f () = (print "hello "; print "world\n")}
\]
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

```ml
exception Oops

exception Ouch of string
```

we have

```ml
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 raise/throw an exception
  - In SML, the keyword is \texttt{raise}
  - For example,
    \begin{verbatim}
    raise Oops
    \end{verbatim}
    or
    \begin{verbatim}
    raise (Ouch "Something broke")
    \end{verbatim}
  - What should the type of \texttt{raise} be?
    \begin{verbatim}
    raise : exn -> 'a
    \end{verbatim}
Exceptions in SML

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

    `<expr> handle <pattern₁> => <handler₁>
    | ... 
    | <patternₙ> => <handlerₙ>

  - 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

```java
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"
```
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 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

  \[
  \text{coins : int list * int } \rightarrow \text{ int list}
  \]
  
  so that, for example,

  \[
  \text{coins ([5,2], 8)}
  \]

  yields

  \[
  [2,2,2,2].
  \]
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

```plaintext
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

```plaintext
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)
```
Adding Exceptions to NQSML

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

\[
e ::= \ldots \mid \text{fail} \mid \text{catch } e_1 \text{ with } e_2
\]
Adding Exceptions to NQSML

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

\[
e ::= \ldots \mid \text{fail} \mid \text{catch } e_1 \text{ with } e_2
\]

Raise the exception

Return value of \( e_1 \) unless it fails, in which case evaluate \( e_2 \)
Static Semantics

\[ \Gamma \otimes \text{fail} : t \]

\[ \Gamma \otimes e_1 : t \quad \Gamma \otimes e_2 : t \]

\[ \Gamma \otimes \text{catch } e_1 \text{ with } e_2 : t \]
Dynamic Semantics

\[
\frac{e_1 \rightarrow e_1'}{
\text{catch } e_1 \text{ with } e_2 \rightarrow \text{catch } e_1' \text{ with } e_2}
\]

\[
\frac{\text{catch } v \text{ with } e_2 \rightarrow v}{\text{catch fail with } e_2 \rightarrow e_2}
\]
Dynamic Semantics

\[
\begin{align*}
  e_1 & \rightarrow e_1' \\
  \text{catch } e_1 \text{ with } e_2 & \rightarrow \text{catch } e_1' \text{ with } e_2 \\
  \text{catch } v \text{ with } e_2 & \rightarrow v \\
  \text{catch } \text{fail} \text{ with } e_2 & \rightarrow e_2
\end{align*}
\]

\[
\begin{align*}
  \text{fail} + e_2 & \rightarrow \text{fail} \\
  v_1 + \text{fail} & \rightarrow \text{fail}
\end{align*}
\]

\[
\begin{align*}
  \text{if fail then } e_2 \text{ else } e_3 & \rightarrow \text{fail}
\end{align*}
\]
Handling Division

\[
\begin{align*}
& e_1 \to e_1' \\
& e_1 \mathbin{\text{div}} e_2 \to e_1' \mathbin{\text{div}} e_2 \\
& e_2 \to e_2' \\
& v_1 \mathbin{\text{div}} e_2 \to v_1 \mathbin{\text{div}} e_2' \\
& n_2 \not\equiv 0 \\
& \overline{n_1} \mathbin{\text{div}} n_2 \to \overline{n_1} \mathbin{\text{div}} n_2 \\
& \overline{n_1} \mathbin{\text{div}} 0 \to \text{fail}
\end{align*}
\]
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:
  if \( e : t \) then either \( e \) is a value or else
  \( e \rightarrow e' \) for some \( e' : t \) or else
  \( e = \text{fail} \).

• Hence if \( e : t \), one of the following is true
  \( e \rightarrow^* v \) (normal termination)
  \( e \rightarrow^* \text{fail} \) (uncaught exception)
  \( e \rightarrow e' \rightarrow e'' \rightarrow e''' \rightarrow ... \) (nontermination)