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 ($expr_1$ `;` $expr_2$)

    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

Example: 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 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 -> 'a
    ```

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"
```

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

```
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)
```

Adding Exceptions to NQSML

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

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

Adding Exceptions to NQSML

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

Raise the exception

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

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

Static Semantics

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

$$\frac{\Gamma \vdash \texttt{e}_1 : \texttt{t} \qquad \Gamma \vdash \texttt{e}_2 : \texttt{t}}{\Gamma \vdash \texttt{catch } \texttt{e}_1 \texttt{ with } \texttt{e}_2 : \texttt{t}}$$

Dynamic Semantics

$$\frac{\mathtt{e_1} \rightarrow \mathtt{e_1'}}{\mathtt{catch\ e_1\ with\ e_2} \text{ ® } \mathtt{catch\ e_1'\ with\ e_2}}$$

$$\frac{}{\mathtt{catch\ v\ with\ e_2} \rightarrow \mathtt{v}}$$

$$\frac{}{\mathtt{catch\ fail\ with\ e_2} \rightarrow \mathtt{e_2}}$$

Dynamic Semantics

$$\frac{\texttt{e}_1 \rightarrow \texttt{e}_1\texttt{'}}{\texttt{catch e}_1\texttt{ with e}_2 \text{ ® } \texttt{catch e}_1\texttt{' with e}_2}$$

$$\frac{}{\texttt{catch v with e}_2 \rightarrow \texttt{v}}$$

$$\frac{}{\texttt{catch fail with e}_2 \rightarrow \texttt{e}_2}$$

$$\frac{}{\texttt{fail + e}_2 \rightarrow \texttt{fail}} \qquad \frac{}{\texttt{v}_1\texttt{ + fail} \rightarrow \texttt{fail}}$$

$$\frac{}{\texttt{if fail then e}_2\texttt{ else e}_3 \rightarrow \texttt{fail}}$$

etc.

Handling Division

$$\frac{\mathtt{e_1} \rightarrow \mathtt{e_1'}}{\mathtt{e_1\ div\ e_2} \rightarrow \mathtt{e_1'\ div\ e_2}}$$

$$\frac{\mathtt{e_2} \rightarrow \mathtt{e_2'}}{\mathtt{v_1\ div\ e_2} \rightarrow \mathtt{v_1\ div\ e_2'}}$$

$$\frac{\underline{\mathtt{n}}_2\ \mathtt{<>}\ \mathtt{0}}{\underline{\mathtt{n}}_1\ \mathtt{div}\ \underline{\mathtt{n}}_2 \rightarrow \underline{\mathtt{n_1\ div\ n_2}}}$$

$$\frac{}{\underline{\mathtt{n}}_1\ \mathtt{div}\ \underline{\mathtt{0}} \rightarrow \mathtt{fail}}$$

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`** $\rightarrow$ **`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`** $\rightarrow$ **`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` $\rightarrow$ **`e' for some e' : t`** *or else*
  `e` = **`fail`**.

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

  `e` $\rightarrow^*$ **`v`** **`(normal termination)`**

  `e` $\rightarrow^*$ **`fail`** **`(uncaught exception)`**

  `e` $\rightarrow$ e' $\rightarrow$ e'' $\rightarrow$ e''' $\rightarrow$ … **`(nontermination)`**