Misc. Announcements

• Assignment available end of the day today
  – Due back in 11/03 (after break)
• Will also update slides on website
  – Today
• Midterm next week
  – Handed out Monday in class
  – Due back 5:00pm Friday

Exceptions

COS 441
Princeton University
Fall 2004

Outline

• Formalize simplified version of exceptions that do not return values
• “Stack unwinding” model of exception handling
• “Two stack” model of exception handling
• Exceptions that return values

Simple Language with Failure

\[
\begin{align*}
\text{Numbers} & \quad n \in \mathbb{N} \\
\text{Types} & \quad \tau ::= \text{int} \\
\text{Expr's} & \quad e ::= n \mid +(e_1,e_2) \\
& \quad \mid \text{fail} \mid \text{try } e_1 \text{ ow } e_2 \\
\text{Values} & \quad v ::= n
\end{align*}
\]

Static Semantics

\[
\begin{align*}
\Gamma \vdash \text{fail} : \tau & \\
\Gamma \vdash e_1 : \tau & \quad \Gamma \vdash e_2 : \tau \\
\Gamma \vdash \text{try } e_1 \text{ ow } e_2 : \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{fail} : \tau & \\
\Gamma \vdash e_1 : \tau & \quad \Gamma \vdash e_2 : \tau \\
\Gamma \vdash \text{try } e_1 \text{ ow } e_2 : \tau
\end{align*}
\]

\[\text{Arbitrary Type}\]
Dynamic Semantics

- Modify C-machine to encode semantics of failure
- M-machine semantics can also be made but messy
  - Similar to adding "error" case for division
  - C-machine formulation allows us to talk about implementation issues in more detail

\[
\begin{align*}
\text{Frames } f & := + (v_1, □) \mid (□, e_2) \\
& \mid \text{try □ ow } e_2
\end{align*}
\]

Type Soundness

**Theorem 13.1 (Preservation)**
If \((K, c)\) ok and \((K, c) \mapsto (K', c')\), then \((K, c)\) ok.

**Proof:** By induction on evaluation.

**Theorem 13.2 (Progress)**
If \((K, c)\) ok then either
1. \(K = \bullet\) and \(c\) value, or
2. \(K = \bullet\) and \(c = \text{fail}\), or
3. there exists \(K'\) and \(c'\) such that \((K, c) \mapsto (K', c')\).

**Proof:** By induction on typing.

Example: Normal Evaluation

\[
\begin{align*}
\text{push handler} & \quad (K, \text{try } e_1 \text{ ow } e_2) \mapsto (\text{try } \square \text{ ow } e_2 \triangleright K, e_1) \\
\text{pop handler} & \quad (\text{try } \square \text{ ow } e_2 \triangleright K, v) \mapsto (K, v) \\
\text{catch} & \quad (\text{try } \square \text{ ow } e_2 \triangleright K, \text{fail}) \mapsto (K, e_2) \\
\text{unwind} & \quad (F \neq \text{try } \square \text{ ow } e_2) \mapsto (K, \text{fail})
\end{align*}
\]

Example: Failure Evaluation

\[
\begin{align*}
\text{push hndl.} & \quad (\bullet, \text{try } e_1(\bullet, 2,\text{ ow } 0)) \\
\text{push} & \quad (\bullet, \text{try } e_1(\bullet, 2,\text{ ow } 0) \triangleright + (1, 2)) \\
\text{push} & \quad (\bullet, \text{try } e_1(\bullet, 2,\text{ ow } 0) \triangleright + (1, 2)) \\
\text{continue} & \quad (\bullet, \text{try } e_1(\bullet, 2,\text{ ow } 0) \triangleright + (1, 2)) \\
\text{pop} & \quad (\bullet, \text{try } e_1(\bullet, 2,\text{ ow } 0) \triangleright + (1, 2)) \\
\text{pop hndl.} & \quad (\bullet, \text{try } e_1(\bullet, 2,\text{ ow } 0) \triangleright + (1, 2)) \\
\end{align*}
\]

Stack Unwinding

- Notice we raise the nearest dynamically enclosing handler
  - Handlers have “dynamic scope”
- Stack unwinding takes a linear number of steps depending on stack depth
  - Not a problem if exceptions are truly exceptional!
  - Bigger problem when we use exceptions to encode other control structures
Non-Exceptional Exceptions

• Common programming trick is to use exceptions to encode non-determinism
  – More on this later
• Used to short circuit deep nested evaluations
  – When checking for list equality fail on first mismatch

List Equality

fun lsteq([],[]) = true
| lsteq(x::xs,y::ys) =
  | (x = y) andalso lsteq(xs,ys)
| lsteq(_,_) = false

Consider two 1000 element list where the last element of the list is different
Code above must pop off 1000 frames to return false

List Equality with Exceptions

exception False
fun lsteq(xs,ys) = let
  fun f([],[]) = ()
  | f(x::xs,y::ys) =
    | if (x = y) then f(xs,ys)
    | else raise False
  | f(_,_) = raise False
  in (f(xs,ys);true) handle False => false
end

List Equality with Exceptions

• If raising exception is a constant time operation version that uses exceptions detects failure faster
• Doesn’t have to unwind or pop stack frames just return with false immediately
• We can design abstract machine which raises exceptions in constant time

Two Stack Approach

Modify C-machine to have two stacks one for pending exception handlers as well as normal control flow

```
Frames f ::= +v1,□ | +□,e2
   | try □ ow e2
Control Stack K ::= • | f ▷ K
Handler Stack H ::= • | (K,e) ▷ H
```

Dynamic Semantics

push handler

\[(H,K,\text{try } e_1 \text{ ow } e_2) \mapsto ((K,e_2) ▷ H,\text{try } □ \text{ ow } e_2 ▷ K,e_1)\]

pop handler

\[((K,e_2) ▷ H,\text{try } □ \text{ ow } e_2 ▷ K,e_1) \mapsto (H,K,e_1)\]

raise handler

\[((K',e') ▷ H,K,\text{fail}) \mapsto (H,K',e')\]

exit

\[•,K,\text{fail}) \mapsto (•,•,\text{fail})\]
Example: Normal Evaluation

\( (\star, \text{try } +((1,2),3) \text{ ow } 0) \mapsto ((\star,0) \triangleright \star, \text{try } \square \text{ ow } 0 \triangleright \star, +((1,2),3)) \)

Example: Failure Evaluation

\( (\star, \text{try } +((1,\text{fail}),3) \text{ ow } 0) \mapsto ((\star,0) \triangleright \star, \text{try } \square \text{ ow } 0 \triangleright \star, +((1,\text{fail}),3)) \)

“Prefix” Property

\( (H, K, \text{try } e_1 \text{ ow } e_2) \mapsto ((K, e_2) \triangleright H, \text{try } \square \text{ ow } e_2 \triangleright K, e_1) \)

- Handler stack captures control stack by saving value of stack pointer
- Prefix property guarantees
  For every state of the form \( ((K', e') \triangleright H', K, e) \)
  \( K = f_0 \triangleright f_1 \triangleright \ldots \triangleright f_n \triangleright K' \)

Low-Level Machine View

Exceptions with Values

Modify our syntax so that exception return values to carry information back

\[
\begin{align*}
\text{Numbers} & \quad n \in \mathbb{N} \\
\text{Types} & \quad \tau ::= \text{int} \mid ?? \\
\text{Exprs} & \quad e ::= n \mid (e_1, e_2) \mid ?? \\
& \quad \quad \quad \mid \text{raise(e)} \mid \text{try e_1 ow e_2} \\
\text{Values} & \quad v ::= n \mid ??
\end{align*}
\]
Type Checking Rules

\[
\begin{align*}
\Gamma & \vdash e : \tau_{\text{exn}} \\
\Gamma & \vdash \text{raise}(e) : \tau \\
\Gamma & \vdash e_1 : \tau \\
\Gamma & \vdash e_2 : \tau_{\text{exn}} \rightarrow \tau \\
\Gamma & \vdash \text{try } e_1 \text{ ow } e_2 : \tau
\end{align*}
\]

Handler is a function accepting exception value

Exceptions Values

Tags \( t \in \ldots \)

Numbers \( n \in \mathbb{N} \)

Types \( \tau ::= \text{int} \mid \text{exn} \)

Expr’s \( e ::= \ldots \mid \mathit{e}_{\text{tag}}(e) \\
\quad \mid \text{dcl} \mathit{exn} \tau(x.e) \\
\quad \mid \text{if} \mathit{tag}(x) \text{ then } (x.e) \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{else } e \)

Values \( v ::= n \mid f(v) \)

Extensible Datatypes

- Exceptions are like Java Object act as universal container for any values
- Exception tags are generate at runtime
- Generative exceptions
  - Arguably a bad idea!
  - Harder to implement not that much extra expressiveness
- Typing rules must be carefully designed to avoid soundness bugs

Puzzle One

```
let
  exception E
val x = E
val y = E
in case (x,y) of
  (E,E) => 1
  (E,_ ) => 2
  (_, E) => 3
  _    => 4
end
```

Puzzle Two

```
let
  exception E
val x = E
val y = E
in case (x,y) of
  (E,E) => 1
  (E,_ ) => 2
  (_, E) => 3
  _    => 4
end
```

Puzzle Three

```
let
  exception E
val x = E
val y = E
in case (x,y) of
  (E,E) => 1
  (E,_ ) => 2
  (_, E) => 3
  _    => 4
end
```
Puzzle One

```
let
  exception E
val x = E
val y = E

in case (x,y) of
  (E,E) => 1 (* Answer *)
  (E,_) => 2
  (_,E) => 3
  _    => 4
end
```

Puzzle Two

```
let
  exception E
val x = E
exception E
val y = E

in case (x,y) of
  (E,E) => 1 (* Answer *)
  (E,_) => 2
  (_,E) => 3
  _    => 4
end
```

Puzzle Three

```
let
  exception E
val x = E
val y = E
exception E

in case (x,y) of
  (E,E) => 1 (* Answer *)
  (E,_) => 2
  (_,E) => 3
  _    => 4
end
```

Summary

- Exceptions allow for non-local exits
- Can be used by programmer for non-exceptional exists not just infrequent errors
- Efficient implementation can be achieved using two stack approach
- Extensible datatypes allow exceptions to carry values