CS6848 - Principles of Programming Languages

Exceptions

V. Krishna Nandivada

IIT Madras

Recap

- Flow analysis using 0-CFA and some simple improvements.
- Closure conversion (revisit).

What you should be able to answer? (necessary not sufficient)

- Given a set of flow constraints solve them to get the flow sets.
- Translate closures in Scheme to C.

Reminder

- Assignment due in 1.5 days.
- Seven more classes to go (Last instructional day for CS6848 - 26th April)
- Final exam on May 1st
- Portion - Post mid-term.

Exceptions

- Real-world programming – a function needs to signal to its caller (or some one in the call chain) that it is not able perform some task. Examples:
  - Division by zero, array out of bounds, out of memory, etc.
- One option is to return a special value. Issue:
  - Every caller has to now look for the special value explicitly.
- Option 2: Automatic transfer of program control. Multiple variants exist:
  - Abort the program when an exception occurs.
  - "throw" the exception – trap + recover (aka "caught")
  - Pass programmer specified data along with the exception – Programmer defined exceptions.

Extending simply typed lambda calculus with errors

- Errors - abort the program.
- Recall: Extending the language requires - extension to syntax, values, type rules and operational semantics.
- Expressions (recall our grammar for lambda calculus)

$$e ::= \cdots | \texttt{error}$$

- Values – we don't add any new values (discussion to follow).
- Types. What should be the type of `error`? Do we need any special types?

Type rules

- There is no restriction on the return type of a function.
- Any function can throw an `error`.
- So for each function $s \to t$, we want the type of $\texttt{error} : t$
- For the program to typecheck:
  - If we allow subtyping: then $\texttt{error} : \bot$.
  - If we allow polymorphism: then $\texttt{error} : \forall X.X$

Operational semantics

- We need rules for only application.

$$\begin{array}{ll} \texttt{error}\ e \to \texttt{error} & AppError1 \\ v\ \texttt{error} \to \texttt{error} & AppError2 \end{array}$$

- Summary: abandon the work if there is an error (during the evaluation of the argument or the function).
- Q: Can we get a situation where we get: `error error` ?
  - NO. Because, `error` is not a value.
- Also note, the evaluation order.

Modification to type soundness

- Recall: Progress lemma: If e is a closed expression, and $A \vdash e : t$ then either e is a value, or there exists e' such that $e \to_V e'$.

We modify it to:

- Recall: Progress lemma: If e is a closed expression, and $A \vdash e : t$ then either e is a value or `error`, or there exists e' such that $e \to_V e'$.

Useless assignment: Prove the type soundness.

Exceptions. Variant 2

- Let us "catch" the exception and do something relevant.
- Extension to syntax

$$e ::= \cdots | \texttt{try}\ e\ \texttt{with}\ e$$

- New typing rules:

$$\text{Type-Try-With}\ \frac{A \vdash e_1 : t \quad A \vdash e_2 : t}{A \vdash \texttt{try}\ e_1\ \texttt{with}\ e_2 : t}$$

Operational semantics

- Evaluating expressions that don't result in `error`.

$$\texttt{try}\ v\ \texttt{with}\ e \rightarrow v$$

- Evaluating an expression that evaluates to an error.

$$\texttt{try error with}\ e \rightarrow e$$

- Step

$$\frac{e_1 \rightarrow e_1'}{\texttt{try}\ e_1\ \texttt{with}\ e_2 \rightarrow \texttt{try}\ e_1'\ \texttt{with}\ e_2}$$

Exceptions variant 3 - User defined

- The program point where the exception is thrown may want to pass information.
- The handler may use this information - to take relevant action (such as recovery, reversal, display some relevant message, and so on).
- Extension to syntax

$$e ::= \cdots | \texttt{throw}\ e\ | \texttt{try}\ e\ \texttt{with}\ e$$

- New typing rules:

$$\text{Type-throw}\ \frac{A \vdash e_1 : t}{A \vdash \texttt{throw}\ e_1\ : t}$$

$$\text{Type-Try-With}\ \frac{A \vdash e_1 : t \qquad A \vdash e_2 : t_1 \rightarrow t}{A \vdash \texttt{try}\ e_1\ \texttt{with}\ e_2 : t}$$

Operational semantics

- Application of a throw.

$$(\texttt{throw}\ v)\ e \rightarrow \texttt{throw}\ v$$

- throw as an argument.

$$v_1\ (\texttt{throw}\ v_2) \rightarrow \texttt{throw}\ v_2$$

- throw of throw

$$\texttt{throw}\ (\texttt{throw}\ v) \rightarrow \texttt{throw}\ v$$

- Step throw.

$$\frac{e_1 \rightarrow e_2}{\texttt{throw}\ e_1 \rightarrow \texttt{throw}\ e_2}$$

Operational semantics (contd)

- try with no exception.

$$\texttt{try}\ v\ \texttt{with}\ e \rightarrow v$$

- Evaluating an expression that throws an expression

$$\texttt{try throw}\ v\ \texttt{with}\ e \rightarrow e\ v$$

- Step try.

$$\frac{e_1 \rightarrow e_1'}{\texttt{try}\ e_1\ \texttt{with}\ e_2 \rightarrow \texttt{try}\ e_1'\ \texttt{with}\ e_2}$$

Recap

- Exceptions
- Reason about programs with exceptions.
- Type rules and operational semantics for languages with exceptions.

Paper reading

- Groups!
- Meet the instructor on Thursday.

Versioning Exceptions

- Traditional exceptions provide only transfer of control.
- Used typically for handling cases when unexpected conditions arise.
- The *store* (maps memory locations to values) is left untouched.
  - It is left to the programmer to manually undo any changes.
  - Q: Is handling the environment (maps variables to values) easy?
- Q: Can we provide transaction semantics to the non-local control flow of control-exceptions?
- Goal: Revert computation to a well-defined state in response to unexpected or undesirable conditions.

Versioning Exceptions

- Each code is protected by an exception handler (installed by `try`.
- A versioned exception ensures that the content of the store, when the exception is raised reflects the program state when the corresponding handler was installed.
- The data generated in the code protected by such exceptions are implicitly versioned.
- Each version is assocated with a particular generative exception value.
- When an exception is raised, the version corresponding to the associated exception value is is restored.
- A handler is provided, which lets the programmer to re-executed the protected code or print error message and so on.

Background needed

- When do you need store?
- Modeling store.

Extending the language with references

Extending the syntax

$$e = \cdots | e; e | ref\ e | !e | e_1 := e_2 | unit$$

- Creating a reference - creates a cell in memory.
- The value stored in the cell is the value the expression e evaluates to.
- Say, r is a reference, then `let` $s = r\ \ e$ makes s an alias to r.
  - Setting $r := 32$, will change the value of s and vice versa.

Extending types

- $t ::= \cdots | Ref\ t | Unit$

Extending values

- $v ::= \cdots | l | unit$

- Think of $Unit$ as the `void` type of C.
- The result of evaluating an expression of type $Unit$ is the constant $unit$.

Type rules

- Reference creation.

$$\frac{A \vdash e : t}{A \vdash ref\ e : Ref\ t}$$

- Dereference

$$\frac{A \vdash e : Ref\ t}{A \vdash !e : t}$$

- Assignment.

$$\frac{A \vdash e_1 : Ref\ t_1 \quad A \vdash e_2 : t_1}{A \vdash e_1 := e_2 : Unit}$$

  - Note: The left hand side is not necessarily a variable.

Modelling the store

- Store can be seen as array of *values*.
- Store can be seen as a map $L \to Values$, where L is the set of locations, and $Values$ is the set of values.
- We use σ to represent the store.
- Rules of operational semantics now will use σ.

Syntax for store

- $$\sigma ::= \Phi | \sigma, l = v$$

Typing store elements

$$\Sigma ::= \Phi | \Sigma, l : t$$

Evaluation rules

Defined over the reflexive, transitive closure of $\to_V$:

$$\to_V : \langle Expression, Store \rangle \to_V \langle Expression, Store \rangle$$

- Step - Application

$$\frac{\langle e_1, \sigma \rangle \to_V \langle e_1', \sigma' \rangle}{\langle e_1 e_2, \sigma \rangle \to_V \langle e_1' e_2, \sigma' \rangle}$$

- Step - Arguments

$$\frac{\langle e_2, \sigma \rangle \to_V \langle e_2', \sigma' \rangle}{\langle v_1 e_2, \sigma \rangle \to_V \langle v_1 e_2' \sigma' \rangle}$$

- Apply

$$\langle (\lambda x.e) v, \sigma \rangle \to_V \langle e[x/v], \sigma \rangle$$

Evaluation rules

- Create reference

$\langle ref\ v, \sigma \rangle \to_V \langle l, \sigma[l \mapsto v] \rangle$, where l is fresh

- Step - reference

$$\frac{\langle e, \sigma \rangle \to_V \langle e', \sigma' \rangle}{\langle ref\ e, \sigma \rangle \to_V \langle ref\ e', \sigma' \rangle}$$

- Dereference a location

$$\langle !l, \sigma \rangle \to_V \langle \sigma(l), \sigma \rangle$$

- Step - Dereference

$$\frac{\langle e, \sigma \rangle \to_V \langle e', \sigma' \rangle}{\langle !e, \sigma \rangle \to_V \langle !e', \sigma' \rangle}$$

Evaluation rules

- Assignment.

$$\langle l := v, \sigma \rangle \to_V \langle unit, \sigma[l \mapsto v] \rangle$$

- Step - Assignment (lhs)

$$\frac{\langle e_1, \sigma \rangle \to_V \langle e_1', \sigma' \rangle}{\langle e_1 := e_2, \sigma \rangle \to_V \langle e_1' := e_2, \sigma' \rangle}$$

- Step - Assignment (rhs)

$$\frac{\langle e_2, \sigma \rangle \to_V \langle e_2', \sigma' \rangle}{\langle l := e_2, \sigma \rangle \to_V \langle l := e_2', \sigma' \rangle}$$

Versioning exceptions

Syntax

- $s \in$ *Simp*::= c | $Pr(x_1, \ldots, x_n)$| ref x | $!x$ | $\lambda\, x.e$ | $x_0(x_1)$ | vExn(x)
  $e \in$ *Exp*::= x| let x=s in e| $x_1 := x_2$| if x then e_1 else e_2| try(y,e)| restore(p,q)
  - $vExn(x)$ – constructs a new exception. x is bound to a procedure that defines the handler for this exception.
  - $try\ (y,e)$ – evaluates y to an exception E, and then evaluates e.
  - $restore\ (p,q)$ – p evaluates to an exception (say E).
    - Raises exception E.
    - Control is transferred to the closest enclosing try expression for E.
    - the handler of E is evaluated with q as the argument.
    - Restores the state.

Q:How to construct try-expression with multiple catches?

Versioning exceptions

Types

- $$\tau ::= \text{Int}\ |Bool|\tau \to \tau|Ref\ \tau|\textbf{Exn}\ (\tau)$$

Extension to type rules

- Exception construction.

$$\frac{A \vdash x : t_1 \to t_2}{A \vdash vExn(x) : Exn(t_1 \to t_2)}$$

- Try block

$$\frac{A \vdash x : Exn(t_1 \to t_2) \qquad A \vdash e : t_2}{A \vdash try(x,e) : t_2}$$

- Restore

$$\frac{A \vdash y : t_1 \qquad A \vdash x : Exn(t_1 \to t_2)}{A \vdash restore(x,y) : t_2}$$

Operational Semantics

CE^2SK machine: Control, Environment (ρ), Exception-stack (Σ), Store (σ), Continuation Pointer (k)

$$\langle x, \rho, k, \sigma, \Sigma \rangle \longrightarrow \langle k, \rho(x), \sigma, \Sigma \rangle$$
$$\langle \text{let}\ x = c\ \text{in}\ e, \rho, k, \sigma, \Sigma \rangle \longrightarrow \langle e, \rho[x \mapsto c], k, \sigma, \Sigma \rangle$$
$$\langle \text{let}\ x = Pr(x_1, \ldots, x_n)\ \text{in}\ e, \rho, k, \sigma, \Sigma \rangle \longrightarrow \langle e, \rho[x \mapsto \texttt{Pr}(\rho(x_1), \rho(x_2), \ldots, \rho(x_n))], k, \sigma, \Sigma \rangle$$
$$\langle \text{let}\ x = \text{ref}\ y\ \text{in}\ e, \rho, k, \sigma, \Sigma \rangle \longrightarrow \langle e, \rho[x \mapsto l], k, \sigma[l \mapsto \rho(y)], \Sigma \rangle \quad \text{for fresh } l$$
$$\langle \text{let}\ x = !y\ \text{in}\ e, \rho, k, \sigma, \Sigma \rangle \longrightarrow \langle e, \rho[x \mapsto \sigma(\rho(y))], k, \sigma, \Sigma \rangle$$
$$\langle \text{let}\ x = \lambda\, y.e'\ \text{in}\ e, \rho, k, \sigma, \Sigma \rangle \longrightarrow \langle e, \rho[x \mapsto \text{clo}\ \langle \lambda\, y.e', \rho \rangle], k, \sigma, \Sigma \rangle$$
$$\langle \text{let}\ x = y(z)\ \text{in}\ e, \rho, k, \sigma, \Sigma \rangle \longrightarrow \langle e', \rho'[w \mapsto \rho(z)], \{\textbf{ret}\ \langle x, e, \rho \rangle\} \oplus k, \sigma, \Sigma \rangle \quad \text{provided } \rho(y) = \text{clo}\ \langle \lambda\, w.e', \rho' \rangle$$
$$\langle x_1 := x_2, \rho, k, \sigma, \Sigma \rangle \longrightarrow \langle k, \rho(x_2), \sigma[\rho(x_1) \mapsto \rho(x_2)], \Sigma \rangle$$
$$\langle \text{if}\ x\ \text{then}\ e_1\ \text{else}\ e_2, \rho, k, \sigma, \Sigma \rangle \longrightarrow \langle e_1, \rho, k, \sigma, \Sigma \rangle \quad \text{provided } \rho(x) = \text{true}$$
$$\langle \text{if}\ x\ \text{then}\ e_1\ \text{else}\ e_2, \rho, k, \sigma, \Sigma \rangle \longrightarrow \langle e_2, \rho, k, \sigma, \Sigma \rangle \quad \text{provided } \rho(x) = \text{false}$$
$$\langle \{\textbf{ret}\ \langle x, e, \rho \rangle\} \oplus k, v, \sigma, \Sigma \rangle \longrightarrow \langle e, \rho[x \mapsto v], k, \sigma, \Sigma \rangle$$

Operational semantics for versioning exceptions (contd)

With exceptions:

$$
\begin{array}{l}
\langle \mathsf{let}\, x = \mathsf{vExn}(y)\, \mathsf{in}\, e, \rho, k, \sigma, \Sigma \rangle \longrightarrow \langle e, \rho[x \mapsto \mathsf{exnVal}\,\langle n, \rho(y)\rangle], k, \sigma, \Sigma \rangle \\
\qquad\qquad\qquad \text{for fresh } n \\
\langle \mathsf{try}(x, e), \rho, k, \sigma, \Sigma \rangle \longrightarrow \langle e, \rho, k, \sigma, \{\langle \rho(x), k, \sigma \rangle\} \oplus \Sigma \rangle \\
\langle \mathsf{restore}(x, y), \rho, k, \sigma, \Sigma \rangle \longrightarrow \langle \{\mathbf{exn}\,\langle n, \rho(y)\rangle\} \oplus k, \sigma, \Sigma \rangle \\
\qquad \text{provided } \rho(x) = \mathsf{exnVal}\,\langle n, v \rangle \\
\langle \{\mathbf{exn}\,\langle n, v\rangle\} \oplus k, \sigma, \Sigma \rangle \longrightarrow \langle e, \rho[x \mapsto v], \{\mathbf{sto}\,\langle \sigma' \rangle\} \oplus k', \sigma, \Sigma'' \rangle \\
\qquad \text{provided } \Sigma = \Sigma' \oplus \{\langle \mathsf{exnVal}\,\langle n, \mathsf{clo}\,\langle \lambda\, x.e, \rho \rangle\rangle, k', \sigma' \rangle\} \oplus \Sigma'' \\
\qquad \text{and } \langle \mathsf{exnVal}\,\langle n, v \rangle, k', \sigma' \rangle \notin \Sigma' \\
\langle \{\mathbf{sto}\,\langle \sigma' \rangle\} \oplus k, v, \sigma, \Sigma \rangle \longrightarrow \langle k, v, \sigma', \Sigma \rangle
\end{array}
$$

- when an exception is thrown, the continuation of the try expression is evaluated in the context of the “versioned” store.
- Changes made to the store in the exception handler are not visible in the continuation.
- Q: What if updates performed in the handler are to be visible in its continuation? – Self reading.

Next class

How to implement Versioning Exceptions.