CSCI 740 - Programming Language Theory

Lecture 17
Types for Imperative Features
Instructor: Hossein Hojjat

October 11, 2017
Imperative Features

- So far we considered types for pure functional languages
- Now we look at types for imperative features
- Use types to characterize side effects
  - references, pointers, assignment, exceptions
- Small step semantics: update heap
ML-Style References

- References to mutable memory cells
- Syntax (as in ML)

\[
e ::= \ldots \mid \text{ref } e \mid e_1 ::= e_2 \mid !e
\]

\[
\tau ::= \ldots \mid \tau \text{ ref}
\]

- Evaluates \(e\), allocates new cell, stores the value of \(e\)
- Returns reference to new cell
  - like \texttt{malloc} + initialization in C, or \texttt{new} in C++ and Java

- \(e_1 ::= e_2\)
  - eval \(e_1\) to a memory cell (reference), updates cell’s value with value of \(e_2\)
    - like \(\ast e_1 = e_2\) in C++

- \(!e\)
  - evaluates \(e\) to a memory cell (reference), returns its contents
    - like \(\ast e\) in C++
Global Effects with Reference Cells

- Cell can escape static scope where created

\[(\lambda f : \text{int} \rightarrow (\text{int ref}). \! (f \ 5)) (\lambda x : \text{int. ref} \ x)\]

- Cell’s value must be visible in whole program
- Result of expression must include changes made to heap
  - (side effects)
- Must extend the evaluation model
Static Semantics of References

\[
\begin{align*}
\Gamma \vdash e : \tau \\
\Gamma \vdash (\text{ref } e) : \tau \text{ ref} \\
\Gamma \vdash e : \tau \text{ ref} \\
\Gamma \vdash !e : \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e_1 : \tau \text{ ref} & \quad \Gamma \vdash e_2 : \tau \\
\Gamma \vdash e_1 := e_2 : \text{unit}
\end{align*}
\]
Heap = mapping from addresses to values

\[ h ::= \emptyset | a \rightarrow \text{val} : \tau \]

- \( a \in \text{Addresses} \)
- Tag heap cells with their types
- Types only for static semantics, not evaluation
  - not a part of the implementation

program = heap + expression

\[ p ::= \text{heap } h \; \text{in} \; e \]

- Initial program: heap \( \emptyset \) in \( e \)
- Heap addresses act as bound variables in the expression
- Allows reuse of local variables properties for heap addresses
  - e.g., we can rename the address and its occurrences
Static Semantics of References

- Typing rules for expressions:

\[ \Gamma \vdash e : \tau \]
\[ \Gamma \vdash \text{ref } e : \tau \text{ ref} \]
\[ \Gamma \vdash \text{ref } e : \tau \text{ ref} \]

- New Rule for programs:

\[ \Gamma \vdash v_i : \tau_i (i = 1 \cdots n) \]
\[ \Gamma \vdash e : \tau \]
\[ \vdash \text{heap } h \text{ in } e : \tau \]

Where

\[ \Gamma = a_1 : \tau_1 \text{ ref}, \cdots, a_n : \tau_n \text{ ref} \]
\[ h = a_1 \rightarrow v_1 : \tau_1, \cdots, a_n \rightarrow v_n : \tau_n \]
Contextual Semantics for References

- New contexts (in addition to the ones before)

\[ H := \text{ref } H \mid H := e \mid a := H \mid !H \]

- No new local reduction rules
- New global reduction rules propagate effects of a write to entire program

\[
\begin{align*}
\text{heap } h \text{ in } H[\text{ref } v : \tau] & \rightarrow \text{heap } h, (a \rightarrow v : \tau) \text{ in } H[a] \\
& \text{(where } a \text{ is fresh)}
\end{align*}
\]

\[
\begin{align*}
\text{heap } h \text{ in } H[!a] & \rightarrow \text{heap } h \text{ in } H[v] \\
& \text{(where } a \rightarrow v : \tau \in h) \\
\end{align*}
\]

\[
\begin{align*}
\text{heap } h \text{ in } H[a := v] & \rightarrow \text{heap } h[a \mapsto v] \text{ in } H[()] \\
& \text{(where } h[a \mapsto v] = h \text{ except cell } a \rightarrow v' : \tau \text{ replaced by } a \rightarrow v : \tau) \\
\end{align*}
\]
Example with References

- Consider the evaluation (redex is underlined)

  \[
  \text{heap } \emptyset \text{ in } (\lambda f : \text{int} \to \text{int ref. } !(f \, 5)) \, (\lambda x : \text{int. } \text{ref } x : \text{int})
  \]

  \[
  \rightarrow \text{heap } \emptyset \text{ in } !(\lambda x : \text{int. } \text{ref } x : \text{int}) \, 5
  \]

  \[
  \rightarrow \text{heap } \emptyset \text{ in } !(\text{ref } 5 : \text{int})
  \]

  \[
  \rightarrow \text{heap } a = 5 : \text{int in } !a
  \]

  \[
  \rightarrow \text{heap } a = 5 : \text{int in } 5
  \]

- Resulting program has a useless memory cell
- No references to it in program
- Equivalent to: \(\text{heap } \emptyset \text{ in } 5\)
- Simple way to model garbage collection
Subtyping References

\[
\frac{\tau_1 \leq \tau_2}{\tau_1 \text{ ref} \leq \tau_2 \text{ ref}}
\]

Unsafe

- Suppose \( \tau_1 \leq \tau_2 \)
- Above rule implies
  \[
x : \tau_2, y : \tau_1 \text{ ref}, f : \tau_1 \rightarrow \text{ int} \vdash y := x; f(!y)
  \]
- **Unsound:** \( f \) called with \( \tau_2 \) but defined only on \( \tau_1 \)
- Java has covariant arrays

References are invariant:

- no subtyping for references
- arrays should be invariant
- mutable records should be invariant
• A mechanism that allows non-local control flow
• Useful for implementing the propagation of errors to caller
• We again use contextual semantics to model exceptions
• Assume that there is a special type \texttt{exn} of exceptions
  • \texttt{exn} could be \texttt{int} to model error codes
• In Java or C++, \texttt{exn} are special object types
Modeling Exceptions

Syntax

\[
e ::= \cdots | \text{raise } e | \text{try } e_1 \text{ handle } x \Rightarrow e_2
\]

\[
\tau ::= \cdots | \text{exn}
\]

- We ignore how exception values are created
  - In examples we use integers as exception values
- Handler binds \( x \) in \( e_2 \) to the actual exception value
- The raised expression propagates to enclosing expressions
- A `raise` expression may appear anywhere
  - “1 + raise 2” is well-typed
  - “if (raise 2) then 1 else 2” is also well-typed
  - “(raise 2) 5” is also well-typed
- What should be the type of `raise`?
Example with Exceptions

- A (strange) factorial function

\[
\text{let } f = \lambda x:\text{int}. \lambda \text{res}:\text{int}. \begin{cases} 
\text{if } x = 0 \text{ then raise res} \\
\text{else } f(x-1)(\text{res} \times x)
\end{cases}
\]

\[
\text{in try } f 5 1 \text{ handle } x \Rightarrow x
\]

- Top-level handler catches the exception and turns it into a regular result
Typing Exceptions

- New typing rules

\[ \frac{\Gamma \vdash e : \text{exn}}{\Gamma \vdash \text{raise } e : \tau} \]

\[ \frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \text{exn} \vdash e_2 : \tau}{\Gamma \vdash \text{try } e_1 \ \text{handle } x \Rightarrow e_2 : \tau} \]

- A `raise` expression has an arbitrary type
  - “`raise e`” can be given any type \( \tau \) that may be required by the context
- The type of the body of `try` and of the handler must match
  - Just like for conditionals
Dynamics of Exceptions

- The result of evaluation can be an uncaught exception
- Evaluation answers:
  
  \[ a ::= v \mid \text{uncaught } v \]

- “uncaught \( v \)” has an arbitrary type
- Raising an exception has global effects
- It is convenient to use contextual semantics
- Exceptions propagate through some contexts but not through others
- We distinguish the handling contexts that intercept exceptions
Contexts for Exceptions

- Contexts

\[ H ::= \circ | H \ e | v \ H | \text{raise } H | \text{try } H \ \text{handle } x \Rightarrow e \]
Contexts for Exceptions

- Contexts

\[ H ::= \circ \mid H \ e \mid v \ H \mid \text{raise} \ H \mid \text{try} \ H \ \text{handle} \ x \Rightarrow e \]

- Can we simply add the following reduction rule?

\[ H[\text{try} \ H'[\text{raise} \ v] \ \text{handle} \ x \Rightarrow e] \rightarrow H[e[x \mapsto v]] \]
• Contexts

\[ H ::= \circ \mid H \ e \mid v \ H \mid \text{raise } H \mid \text{try } H \ \text{handle } x \Rightarrow e \]

• Can we simply add the following reduction rule?

\[ H[\text{try } H'[\text{raise } v] \ \text{handle } x \Rightarrow e] \rightarrow H[e[x \mapsto v]] \]

• This requires a side condition:

\( H' \) does not contain a handler for the exception!
Contexts for Exceptions

- **Contexts**
  \[ H ::= \circ \mid H \ e \mid \nu \ H \mid \text{raise} \ H \mid \text{try} \ H \ \text{handle} \ x \Rightarrow e \]

- **Propagating contexts:**
  Contexts that propagate exceptions to their own enclosing contexts
  \[ P ::= \circ \mid P \ e \mid \nu \ P \mid \text{raise} \ P \]

**Decomposition theorem**

If \( e \) is not a value and \( e \) is well-typed then it can be decomposed in exactly one of the following ways:

- \( H[\lambda x : \tau.e \ \nu] \) (normal lambda calculus)
- \( H[\text{try} \ \nu \ \text{handle} \ x \Rightarrow e] \) (handle it or not)
- \( H[\text{try} \ P[\text{raise} \ \nu] \ \text{handle} \ x \Rightarrow e] \) (propagate!)
- \( P[\text{raise} \ \nu] \) (uncaught exception)
Contextual Semantics for Exceptions

- Small-step reduction rules

\[ H[(\lambda x: \tau. e) v] \rightarrow H[e[x \mapsto v]] \]
\[ H[\text{try } v \text{ handle } x \Rightarrow e] \rightarrow H[v] \]
\[ H[\text{try } P[\text{raise } v] \text{ handle } x \Rightarrow e] \rightarrow H[e[x \mapsto v]] \]
\[ P[\text{raise } v] \rightarrow \text{uncaught } v \]

- The handler is ignored if the body of try completes normally
- A raised exception propagates (in one step) to the closest enclosing handler or to the top of the program