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)

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

$$\tau ::= \ldots \mid \tau \text{ ref}$$

ref e

- Evaluates e, allocates new cell, stores the value of e
- Returns reference to new cell
  - like `malloc` + initialization in C, or 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 `*e1 = e2` in C++

$!e$

- evaluates e to a memory cell (reference), returns its contents
  - like `*e` in C++

Global Effects with Reference Cells

- Cell can escape static scope where created

$$\big(\lambda f : \text{int} \to (\text{int ref}).\ !(f\ 5)\big)\big(\lambda x : \text{int}.\ \text{ref}\ x\big)$$

- 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

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\textsf{ref}\ e) : \tau\ \textsf{ref}}$$

$$\frac{\Gamma \vdash e : \tau\ \textsf{ref}}{\Gamma \vdash\ !e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \textsf{ref} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash\ e_1 := e_2 :\ \textsf{unit}}$$

Modeling References

Heap = mapping from addresses to values

$$h := \emptyset \mid , a \rightarrow \textit{val} : \tau$$

- $a \in$ 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:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textsf{ref}\ e : \tau\ \textsf{ref}} \qquad \frac{\Gamma \vdash e :\ \textsf{ref}}{\Gamma \vdash\ !e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau\ \textsf{ref} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash\ e_1 := e_2 :\ \textsf{unit}}$$

- New Rule for programs:

$$\frac{\Gamma \vdash v_i : \tau_i (i = 1 \cdots n) \qquad \Gamma \vdash e : \tau}{\vdash\ \textsf{heap}\ h\ \textsf{in}\ e : \tau}$$

Where

$$\Gamma = a_1 : \tau_1\ \textsf{ref}, \cdots, a_n : \tau_n\ \textsf{ref}$$
$$h = a_1 \to v_1 : \tau_1, \cdots, a_n \to v_n : \tau_n$$

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

$$\boxed{\text{heap } h \text{ in } H[\text{ref } v : \tau] \to \text{heap } h, (a \to v : \tau) \text{ in } H[a]}$$

(where a is fresh)

$$\boxed{\text{heap } h \text{ in } H[!a] \to \text{ heap } h \text{ in } H[v]}$$

(where $a \to v : \tau \in h$)

$$\boxed{\text{heap } h \text{ in } H[a := v] \to \text{ heap } h[a \mapsto v] \text{ in } H[()]}$$

(where $h[a \mapsto v] = h$ except cell $a \to v' : \tau$ replaced by $a \to v : \tau$)

Example with References

- Consider the evaluation (redex is underlined)

$$
\begin{aligned}
& \text{heap } \emptyset \text{ in } \underline{(\lambda f : \text{int} \to \text{int ref}.\ !(f\ 5))\,(\lambda x : \text{int}.\ \text{ref } x : \text{int})} \\
\to\ & \text{heap } \emptyset \text{ in } \underline{!((\lambda x : \text{int}.\ \text{ref } x : \text{int})\ 5)} \\
\to\ & \underline{\text{heap } \emptyset \text{ in } !(\text{ref } 5 : \text{int})} \\
\to\ & \underline{\text{heap } a = 5 : \text{int in } !a} \\
\to\ & \text{heap } a = 5 : \text{int in } 5
\end{aligned}
$$

- Resulting program has a useless memory cell
- No references to it in program
- Equivalent to: heap $\emptyset$ 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 τ_2 but defined only on τ_1
- Java has covariant arrays

References are invariant:

- no subtyping for references
- arrays should be invariant
- mutable records should be invariant

Exceptions

- 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 exn of exceptions
    - exn could be int to model error codes
- In Java or C++, exn are special object types

Modeling Exceptions

Syntax

$$e ::= \cdots \mid \texttt{raise}\ e \mid \texttt{try}\ e_1\ \texttt{handle}\ x \Rightarrow e_2$$

$$\tau ::= \cdots \mid \mathsf{exn}$$

- 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
  - "$1 + \texttt{raise}\ 2$" is well-typed
  - "if $(\texttt{raise}\ 2)$ then 1 else 2" is also well-typed
  - "$(\texttt{raise}\ 2)\ 5$" is also well-typed
- What should be the type of `raise`?

Example with Exceptions

- A (strange) factorial function

```
let f = λ x:int. λ res:int. if x = 0 then raise res
   else f (x − 1) (res × x)
      in try f 5 1 handle x ⇒ x
```

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

Typing Exceptions

- New typing rules

$$\frac{\Gamma \vdash e : \mathsf{exn}}{\Gamma \vdash \texttt{raise}\ e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma, x : \mathsf{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{handle}\ x \Rightarrow e_2 : \tau}$$

- A `raise` expression has an arbitrary type
    - "`raise` e" can be given any type τ 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:

$$a ::= v \mid \textsf{uncaught } v$$

- "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

$$H ::= \circ \mid H\ e \mid v\ H \mid \texttt{raise}\ H \mid \texttt{try}\ H\ \texttt{handle}\ x \Rightarrow e$$

Contexts for Exceptions

- Contexts

$$H ::= \circ \mid H\ e \mid v\ H \mid \texttt{raise}\ H \mid \texttt{try}\ H\ \texttt{handle}\ x \Rightarrow e$$

- Can we simply add the following reduction rule?

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

Contexts for Exceptions

- Contexts

$$H ::= \circ \mid H\ e \mid v\ H \mid \texttt{raise}\ H \mid \texttt{try}\ H\ \texttt{handle}\ x \Rightarrow e$$

- Can we simply add the following reduction rule?

$$H[\texttt{try}\ H'[\texttt{raise}\ v]\ \texttt{handle}\ x \Rightarrow e] \to H[e[x \mapsto v]]$$

- This requires a side condition:
  H' does not contain a handler for the exception!

Contexts for Exceptions

- Contexts

$$H ::= \circ \mid H\ e \mid v\ H \mid \texttt{raise}\ H \mid \texttt{try}\ H\ \texttt{handle}\ x \Rightarrow e$$

- **Propagating contexts:**
  Contexts that propagate exceptions to their own enclosing contexts

$$P ::= \circ \mid P\ e \mid v\ P \mid \texttt{raise}\ P$$

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:

- $H[(\lambda x : \tau.e)\ v]$ (normal lambda calculus)
- $H[\texttt{try}\ v\ \texttt{handle}\ x \Rightarrow e]$ (handle it or not)
- $H[\texttt{try}\ P[\texttt{raise}\ v]\ \texttt{handle}\ x \Rightarrow e]$ (propagate!)
- $P[\texttt{raise}\ v]$ (uncaught exception)

Contextual Semantics for Exceptions

- Small-step reduction rules

$$
\begin{array}{lcl}
H[(\lambda x:\tau.e)\ v] & \to & H[e[x \mapsto v]] \\
H[\texttt{try}\ v\ \texttt{handle}\ x \Rightarrow e] & \to & H[v] \\
H[\texttt{try}\ P[\texttt{raise}\ v]\ \texttt{handle}\ x \Rightarrow e] & \to & H[e[x \mapsto v]] \\
P[\texttt{raise}\ v] & \to & \text{uncaught}\ v
\end{array}
$$

- 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