Elements of Programming Languages
Lecture 14: References, Arrays, and Resources

James Cheney
University of Edinburgh
November 17, 2015

0. Over the final few lectures we are exploring cross-cutting design issues
0. Today we consider a way to incorporate mutable variables/assignment into a functional setting:
   - References
   - Interaction with subtyping and polymorphism
   - Resources, more generally

0. In LWhile, all variables are mutable and global
0. This makes programming fairly tedious and it’s easy to make mistakes
0. There’s also no way to create new variables (short of coming up with a new variable name)
0. Can we smoothly add mutable state side-effects to LPoly?
0. Can we provide imperative features within a mostly-functional language?

0. Consider the following language LRef extending LPoly:

\[
\begin{align*}
  e & ::= \cdots \mid \text{ref}(e) \mid !e \mid e_1 := e_2 \mid e_1 ; e_2 \\
  \tau & ::= \cdots \mid \text{ref}[\tau]
\end{align*}
\]

0. Idea: \text{ref}(e) evaluates e to v and creates a new reference cell containing v
0. !e evaluates e to a reference and looks up its value
0. \(e_1 := e_2\) evaluates \(e_1\) to a reference cell and \(e_2\) to a value and assigns the value to the reference cell.
0. \(e_1 ; e_2\) evaluates \(e_1\), ignores value, then evaluates \(e_2\)
References: Types

\[
\begin{array}{c}
\Gamma \vdash e : \tau \\
\hline
\Gamma \vdash e : \tau \\
\hline
\Gamma \vdash \text{ref}(e) : \text{ref}[\tau] \\
\hline
\Gamma \vdash e : \text{ref}[\tau] \\
\hline
\Gamma \vdash e : \text{ref}[\tau] \\
\hline
\Gamma \vdash e_1 : \text{unit} \\
\hline
\Gamma \vdash e_2 : \tau \\
\hline
\Gamma \vdash e_1 := e_2 : \text{unit} \\
\end{array}
\]

- ref(e) creates a reference of type \( \tau \) if \( e : \tau \)
- !e gets a value of type \( \tau \) if \( e : \text{ref}[\tau] \)
- \( e_1 := e_2 \) updates reference \( e_1 : \text{ref}[\tau] \) with value \( e_2 : \tau \).
- Its return value is ()
- \( e_1; e_2 \) evaluates \( e_1 \), ignores the resulting (), and evaluates \( e_2 \).

Interpreting references in Scala using Ref

```scala
case class Ref(e: Expr) extends Expr
case class Deref(e: Expr) extends Expr
case class Assign(e1, e2: Expr) extends Expr
case class Cell(l: Ref[Value]) extends Value

def eval(env: Env[Value], e: Expr) = e match {
  case Ref(e) => Cell(new Ref(eval(env,e)))
  case Deref(e) => eval(env,e) match {
    case Cell(r) => r.get
  }
  case Assign(e1,e2) => eval(env,e1) match {
    case Cell(r) => r.set(eval(env,e2))
  }
  case _ => ()
}
```

Imperative Programming and Procedures

- Once we add references to a functional language (e.g. L_Poly), we can use function definitions and lambda-abstraction to define procedures
- Basically, a procedure is just a function with return type unit

```scala
val x = new Ref(42)
def incrBy(n: Int): () = {
  x.set(x.get + n)
}
```

- Such a procedure does not return a value, and is only executed for its “side effects” on references
- Using the same idea, we can embed all of the constructs of LWhile in LRef (see tutorial)
References: Semantics

- Small steps $\sigma, e \mapsto \sigma', e'$, where $\sigma : \text{Loc} \rightarrow \text{Value}$. "in initial state $\sigma$, expression $e$ can step to $e'$ with state $\sigma'"$.
- What does $\text{ref}(e)$ evaluate to? A pointer or memory cell location, $\ell \in \text{Loc}$

$$\nu ::= \ldots | \ell$$

- These special values only appear during evaluation.

$[\sigma, e \mapsto \sigma', e']$ for $L_{\text{Ref}}$

$$\frac{\ell \notin \text{locs}(\sigma)}{\sigma, \text{ref}(\nu) \mapsto \sigma[\ell := \nu], \ell}$$

$\sigma, !\ell \mapsto \sigma, \sigma(\ell)$

$\sigma, \ell := \nu \mapsto \sigma[\ell := \nu], ()$

References: Semantics

- Finally, we need rules that evaluate inside the reference constructs themselves:

$[\sigma, e \mapsto \sigma', e']$

$$\frac{\sigma, e \mapsto \sigma', e'}{\sigma, \text{ref}(e) \mapsto \sigma', \text{ref}(e')}$$

$$\frac{\sigma, e \mapsto \sigma', e'}{\sigma, !e \mapsto \sigma', !e'}$$

$$\frac{\sigma, e_1 \mapsto \sigma', e_1'}{\sigma, e_1 := e_2 \mapsto \sigma', e_1' := e_2}$$

$$\frac{\sigma, e_2 \mapsto \sigma', e_2'}{\sigma, v_1 := e_2 \mapsto \sigma', v_1 := e_2'}$$

- Notice again that we need to allow for updates to $\sigma$.
- For example, to evaluate $\text{ref}(\text{ref}(42))$

References: Examples

- Simple example

$$\text{let } r = \text{ref}(42) \text{ in } r := 17; !r \mapsto [\ell := 42], \text{let } r = \ell \text{ in } r := 17; !r$$

$$\mapsto [\ell := 42], \ell := 17; !\ell \mapsto [\ell := 17], 17$$

- Aliasing/copying

$$\text{let } r = \text{ref}(42) \text{ in } (\lambda x. \lambda y. x := !y + 1) r r \mapsto [\ell := 42], \text{let } r = \ell \text{ in } (\lambda x. \lambda y. x := !y + 1) r r$$

$$\mapsto [\ell := 42], (\lambda x. \lambda y. x := !y + 1) \ell \ell \mapsto [\ell := 42], (\lambda y. !\ell := y + 1) \ell$$

$$\mapsto [\ell := 42], \ell := !\ell + 1 \mapsto [\ell := 42], \ell := 42 + 1$$

$$\mapsto [\ell := 42], \ell := 43 \mapsto [\ell := 43], ()$$
Something’s missing

- We didn’t give a rule for \( e_1; e_2 \). It’s pretty straightforward (exercise!)
- actually, \( e_1; e_2 \) is definable as

\[
e_1; e_2 \iff \text{let } _= e_1 \text{ in } e_2
\]

where \( _= \) stands for any variable not already in use in \( e_1, e_2 \).

- Why?
  - To evaluate \( e_1; e_2 \), we evaluate \( e_1 \) for its side effects, ignore the result, and then evaluate \( e_2 \) for its value (plus any side effects)
  - Evaluating \( \text{let } _= e_1 \text{ in } e_2 \) first evaluates \( e_1 \), then binds the resulting \( () \) to some variable not used in \( e_2 \), and finally evaluates \( e_2 \).

Reference semantics: observations

- Notice that any subexpression can create, read or assign a reference:

\[
\text{let } r = \text{ref}(1) \text{ in } (r := 1000; 3) + !r
\]

- This means that evaluation order really matters!
- Do we get 4 or 1003 from the above?
  - With left-to-right order, \( r := 1000 \) is evaluated first, then \( !r \), so we get 1003
  - If we evaluated right-to-left, then \( !r \) would evaluate to 1, before assigning \( r := 1000 \), so we would get 4
- However, the small-step rules clarify that existing constructs evaluate “as usual”, with no side-effects.

Arrays

- Arrays generalize references to allow getting and setting by index (i.e. a reference is a one-element array)

\[
e ::= \cdots | \text{array}(e_1, e_2) | e_1[e_2] | e_1[e_2] := e_3
\]

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

- \( \text{array}(n, \text{init}) \) creates an array of \( n \) elements, initialized to \( \text{init} \)
- \( \text{arr}[i] \) gets the \( i \)th element; \( \text{arr}[i] := v \) sets the \( i \)th element to \( v \)
- This introduces the potential problem of out-of-bounds accesses
- Typing, evaluation rules for arrays: exercise

References and subtyping

- Suppose we have an abstract class \( C \) with subclass \( D \).
- Suppose we allowed contravariant subtyping for \( \text{Ref} \), i.e. \( \text{Ref}[-A] \)
- We could then do the following:

\[
\begin{align*}
\text{val } x: \text{Ref}[C] &= \text{new Ref(new} \ C(...)) \\
// D <: C, hence \text{Ref}[C] <: \text{Ref}[D] \\
x.\text{callDOnlyFunction}(...) &// \text{unsound!}
\end{align*}
\]

- which is obviously silly: we shouldn’t expect a reference to \( C \) to be castable to \( D \).
References and subtyping

- Suppose we have an abstract class C with subclass D.
- Suppose we allowed covariant subtyping for Ref, i.e.: Ref[+A]
- We could then do the following:

```plaintext
val x: Ref[D] = new Ref(new D(...))
// D <: C, so Ref[C] => Unit <: Ref[D] => Unit
x.set(new C(...)) // x still has type Ref[D]
```

Therefore, mutable parameterized types like Ref must be invariant (neither covariant nor contravariant)

- (Java got this wrong, for built-in array types!)

References and polymorphism [non-examinable]

- A related problem: references can violate type soundness in a language with Hindley-Milner style type inference and let-bound polymorphism (e.g. ML, OCaml, F#)
  
  let r = ref (fn x => x) in
  r := (fn x => x + 1);
  !r(true)

- We then assign r to be a function of type int → int
  and then apply r to a boolean!
- Accepted solution: the value restriction - the right-hand side of a polymorphic let must be a value.

- (e.g., in Scala, polymorphism is only introduced via function definitions)

Resources

- References, arrays illustrate a common resource pattern:
  - Memory cells (references, arrays, etc.)
  - Files/file handles
  - Database, network connections
  - Locks
- Usage pattern: allocate/open/acquire, use, deallocate/close/release
- Key issues:
  - How to ensure proper use?
  - How to ensure eventual deallocation?
  - How to avoid attempted use after deallocation?
- Some languages (notably C/C++) distinguish between type τ and type τ* (“pointer to τ”), i.e. a mutable reference
- Other languages, notably Java, consider many types (e.g. classes) to be “reference types”, i.e., all variables of that type are really mutable (and nullable!) references.
- In Scala, variables introduced by val are immutable, while using var they can be assigned.
- In Haskell, as a pure, functional language, all variables are immutable; references and mutable state are available but must be handled specially
Safe allocation and use of resources

- In a strongly typed language, we can ensure safe resource use by ensuring all expressions of type `ref[τ]` are properly initialized.
- C/C++ does not do this: a pointer `τ*` may be “uninitialized” (not point to an allocated `τ` block). Must be initialized separately via `malloc` or other operations.
- Java (sort of) does this: an expression of reference type `τ` is a reference to an allocated `τ` (or null!)?
- Scala, Haskell don’t allow “silent” null values, and so a `τ` is always an allocated structure?
- Moreover, a `ref[τ]` is always a reference to an allocated, mutable `τ`.

Safe deallocation of resources?

- Unfortunately, types are not as helpful in enforcing safe deallocation.
- One problem: forgetting to deallocate (resource leaks). Leads to poor performance or run-time failure if resources exhausted.
- Another problem: deallocating the same resource more than once (double free), or trying to use it after it’s been deallocated.
- A major reason is aliasing: copies of references to allocated resources can propagate to unpredictable parts of the program.
- Substructural typing discipline (guest lecture 2) can help with this, but remains an active research topic...

Main approaches to deallocation

- C/C++: explicit deallocation (`free`) must be done by the programmer.
  - (This is very very hard to get right.)
- Java, Scala, Haskell use garbage collection. It is the runtime’s job to decide when it is safe to deallocate resources.
  - This makes life much easier for the programmer, but requires a much more sophisticated implementation, and complicates optimization/performance tuning.
- Lexical scoping or exception handling works well for ensuring deallocation in certain common cases (e.g. files, locks, connections).
- Other approaches include reference counting, regions, etc.

Summary

- We continued to explore design considerations that affect many aspects of a language.
- Today:
  - references and mutability, in generality
  - interaction with subtyping and polymorphism
  - some observations about other forms of resources and the “allocate/use/deallocate” pattern.