Overview

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

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

James Cheney
University of Edinburgh
November 13, 2017

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

Consider the following language L_{Ref} extending L_{Poly}:

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

Idea: ref(e) evaluates e to v and creates a new reference cell containing v
!e evaluates e to a reference and looks up its value
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.
e_1; e_2 evaluates e_1, ignores value, then evaluates e_2
## References: Types

<table>
<thead>
<tr>
<th>( \Gamma \vdash e : \tau ) for ( L_{Ref} )</th>
</tr>
</thead>
</table>
| \( \Gamma \vdash e : \tau \) \[ \Gamma \vdash e : \text{ref}[\tau] \]
| \( \Gamma \vdash e_1 : \text{ref}[\tau] \quad \Gamma \vdash e_2 : \tau \) \[ \Gamma \vdash e_1 := e_2 : \text{unit} \]

- \( \text{ref}(e) \) 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 \( e_2 : \tau \).
- \( e_1 ; e_2 \) evaluates \( e_1 \), ignores the resulting value, and evaluates \( e_2 \).

### References in Scala

Recall that `var` in Scala makes a variable mutable:

```scala
class Ref[A](val x: A) {
  private var a = x
  def get = a
  def set(y: A) = { a = y }
}
```

```scala
global x = new Ref[Int](1)
global x.set(12)
scala> x.get
res5: Int = 12
```

### 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 Cell(r) => r.get
}
```

### 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
global x = new Ref(42)
global incrBy(n: Int): Unit = { 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 \( L_{While} \) in \( L_{Ref} \) (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, $\ell \in \text{Loc}$

$$ v ::= \cdots | \ell $$

- These special values only appear during evaluation.

**$[\sigma, e \mapsto \sigma', e']$ for $\text{LRef}$**

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

References: Semantics

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

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

\[ \frac{\sigma, e \mapsto \sigma', e'}{[\sigma, \text{ref}(e) \mapsto \sigma', \text{ref}(e')]}, \quad [\sigma, e \mapsto \sigma', e'] \]
\[ \frac{\sigma, e_1 \mapsto \sigma', e'_1}{\sigma, e_1 := e_2 \mapsto \sigma', e'_1 := e_2}, \quad \sigma, v_1 := e_2 \mapsto \sigma', v_1 := e'_2} \]

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

**References: Semantics**

- We also need to change all of the existing small-step rules to pass $\sigma$ through...

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

\[ \sigma, e_1 \mapsto \sigma', e'_1 \]
\[ \sigma, v_1 + v_2 \mapsto \sigma, v_1 + v_2 \]
\[ \sigma, v_1 \times v_2 \mapsto \sigma, v_1 \times v_2 \]

- Subexpressions may contain references (leading to allocation or updates), so we need to allow $\sigma$ to change in any subexpression evaluation step.

References: Examples

- Simple example

```
let r = \text{ref}(42) in r := 17; r
```

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

- Aliasing/copying

```
let r = \text{ref}(42) in (\lambda x. \lambda y.x := !y + 1) r r
```

\[ [\ell := 42], \text{let } r = \ell \text{ in } (\lambda x. \lambda y.x := !y + 1) r r \]
\[ [\ell := 42], (\lambda x. \lambda y.x := !y + 1) \ell \ell \]
\[ [\ell := 42], (\lambda y. \ell := !y + 1) \ell \]
\[ [\ell := 42], \ell := !\ell + 1 \mapsto [\ell := 42], \ell := 42 + 1 \]
\[ [\ell := 42], \ell := 43 \mapsto [\ell := 43], () \]
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 value 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}) \text{ creates an array of } n \text{ elements, initialized to } \text{init}
  \\
  \text{arr}[i] \text{ gets the } i\text{th element; } \text{arr}[i] := v \text{ sets the } i\text{th element to } v
  \\
  \text{This introduces the potential problem of out-of-bounds accesses}
  \\
  \text{Typing, evaluation rules for arrays: exercise}
  
  
References and subtyping

- Consider Integer < Object, String < Object
- Suppose we allowed contravariant subtyping for \( \text{Ref} \), i.e. \( \text{Ref}[-A] \)
  which is obviously silly: we shouldn’t expect a reference to Object to be castable to String.
- We could then do the following:
  \[
  \text{val } x: \text{Ref}[\text{Object}] = \text{new } \text{Ref}(\text{new Integer}(42))
  // String < Object,
  // hence \text{Ref}[\text{Object}] <: \text{Ref}[\text{String}]
  x.\text{get}\text{.length}() \text{ // unsound! } x: \text{Ref}[\text{Int}]
Consider Int <: Object, String <: Object
Suppose we allowed covariant subtyping for Ref, i.e.
Ref[+A]
We could then do the following:

```scala
case class Ref[A]
val x: Ref[String] = new Ref(new String("asdf"))
def bad(y: Ref[Object]) = y.set(new Integer(42))
bad(x) // x still has type Ref[String]!
x.get.length() // unsound!
```

Therefore, mutable parameterized types like Ref must be invariant (neither covariant nor contravariant)
(Java got this wrong, for built-in array types!)

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#)

```scala
let r = ref (fn x => x) in
r := (fn x => x + 1);
!r(true)
```

r initially gets inferred type \( \forall A . A \rightarrow A \)
We then assign r to be a function of type int \( \rightarrow \) 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)

References and polymorphism [non-examinable]

Some languages (notably C/C++) distinguish between type \( \tau \) and type \( \tau \ast \) ("pointer to \( \tau \)"), 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 usable (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`.

### Main approaches to deallocation

- C/C++: explicit deallocation (**free**) must be done by the programmer.
  - (This is very hard to get right, and causes many bugs.)
- 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 general
  - interaction with subtyping
  - and polymorphism [non-examinable]
  - some observations about other forms of resources and the “allocate/use/deallocate” pattern.