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
In $L_{\text{While}}$, 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 $L_{\text{Poly}}$?

Can we provide imperative features within a mostly-functional language?
Consider the following language $L_{\text{Ref}}$ extending $L_{\text{Poly}}$:

$$
e ::= \cdots | \text{ref}(e) | !e | e_1 := e_2 | e_1; e_2$$  
$$\tau ::= \cdots | \text{ref}[\tau]$$

Idea: \text{ref}(e) evaluates $e$ to $v$ and creates a \textbf{new reference cell} containing $v$

!e evaluates $e$ to a reference and \textbf{looks up its value}

e_1 := e_2 evaluates $e_1$ to a reference cell and $e_2$ to a value and \textbf{assigns} the value to the reference cell.

e_1; e_2 evaluates $e_1$, ignores value, then evaluates $e_2$
References: Types

\[ \Gamma \vdash e : \tau \] for L_{Ref}

- \( \Gamma \vdash e : \tau \) \implies \( \Gamma \vdash \text{ref}(e) : \text{ref}[\tau] \)
- \( \Gamma \vdash e : \text{ref}[\tau] \) \implies \( \Gamma \vdash e_1 := e_2 : \text{unit} \)
- \( \Gamma \vdash e : \text{ref}[\tau] \) \implies \( \Gamma \vdash e_1 : \text{ref}[\tau] \)
- \( \Gamma \vdash e : \text{ref}[\tau] \) \implies \( \Gamma \vdash e_2 : \tau \)
- \( \Gamma \vdash e_1 : \tau' \) \implies \( \Gamma \vdash e_2 : \tau' \)
- \( \Gamma \vdash e_1 ; e_2 : \tau \)

- \( \text{ref}(e) \) 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 \( e_2 : \tau \). Its return value is ()
- \( e_1 ; e_2 \) evaluates \( e_1 \), ignores the resulting value, and evaluates \( e_2 \).
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> val x = new Ref[Int](1)
x: Ref[Int] = Ref@725bef66
scala> x.get
res3: Int = 1
scala> x.set(12)
scala> x.get
res5: Int = 12
```
Interpreting references in Scala using Ref

You are to implement this in Assignment 3!
Imperative Programming and Procedures

- Once we add references to a functional language (e.g. L\textsubscript{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): 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\textsubscript{While} in L\textsubscript{Ref} (see tutorial).
References: Semantics

- Small steps \(\sigma, e \mapsto \sigma', e'\), where \(\sigma : \text{Loc} \to \text{Value}\). “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.

![Box with equations]

\(\sigma, e \mapsto \sigma', e'\) for \(\text{L}_{\text{Ref}}\)

\[
\begin{align*}
\ell \notin \text{locs}(\sigma) \\
\sigma, \text{ref}(v) \mapsto \sigma[\ell := v], \ell
\end{align*}
\]

\[
\begin{align*}
\sigma, !\ell & \mapsto \sigma, \sigma(\ell) \\
\sigma, \ell := v & \mapsto \sigma[\ell := v], ()
\end{align*}
\]
We also need to change all of the existing small-step rules to pass $\sigma$ through...

$\sigma, e \mapsto \sigma', e'$

\[
\begin{align*}
\sigma, e_1 & \mapsto \sigma', e'_1 \\
\sigma, e_1 \oplus e_2 & \mapsto \sigma', e'_1 \oplus e_2 \\
\sigma, v_1 \perp v_2 & \mapsto \sigma, v_1 \perp_N v_2 \\
\sigma, e_2 & \mapsto \sigma', e'_2 \\
\sigma, v_1 \oplus e_2 & \mapsto \sigma', v_1 \oplus e'_2 \\
\sigma, v_1 \times v_2 & \mapsto \sigma, v_1 \times_N v_2 \\
\vdots
\end{align*}
\]

Subexpressions may contain references (leading to allocation or updates), so we need to allow $\sigma$ to change in any subexpression evaluation step.
Finally, we need rules that evaluate inside the reference constructs themselves:

\[
\begin{align*}
\sigma, e & \mapsto \sigma', e' \\
\sigma, \text{ref}(e) & \mapsto \sigma', \text{ref}(e') \\
\sigma, e_1 & \mapsto \sigma', e'_1 \\
\sigma, e_1 := e_2 & \mapsto \sigma', e'_1 := e_2 \\
\sigma, v_1 & \mapsto \sigma', v_1 := e'_2
\end{align*}
\]

Notice again that we need to allow for updates to \(\sigma\).

For example, to evaluate \(\text{ref}(\text{ref}(42))\)
References: Examples

- Simple example

```latex
let r = ref(42) 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], !\ell \mapsto [\ell := 17], 17
```
References: Examples

- **Simple example**

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

- **Aliasing/copying**

  \[
  \text{let } r = \text{ref}(42) \text{ in } (\lambda x. \lambda y. x := !y + 1) \ r \ r \\
  \rightarrow [\ell = 42], \text{let } r = \ell \text{ in } (\lambda x. \lambda y. x := !y + 1) \ r \ r \\
  \rightarrow [\ell = 42], (\lambda x. \lambda y. x := !y + 1) \ \ell \ \ell \\
  \rightarrow [\ell = 42], (\lambda y. \ell := !y + 1) \ \ell \\
  \rightarrow [\ell = 42], \ell := !\ell + 1 \mapsto [\ell = 42], \ell := 42 + 1 \\
  \rightarrow [\ell = 42], \ell := 43 \mapsto [\ell = 43], ()
  \]
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)

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

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

- \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] := \nu sets the \( i \)th element to \( \nu \)

- This introduces the potential problem of *out-of-bounds* accesses

- Typing, evaluation rules for arrays: exercise
References and subtyping

- Consider Integer <: Object, String <: Object
- Suppose we allowed *contravariant* subtyping for Ref, i.e. 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:

```scala
val x: Ref[Object] = new Ref(new Integer(42))
// String <: Object,
// hence Ref[Object] <: Ref[String]
x.get.length() // unsound! x: Ref[Int]
```
References and subtyping

- Consider `Int <: Object, String <: Object`
- Suppose we allowed *covariant* subtyping for `Ref`, i.e. `Ref [+A]`
- We could then do the following:

```scala
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!)
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#)

```ml
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)
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? (e.g. all array accesses are in-bounds)
  - How to ensure eventual deallocation?
  - How to avoid attempted use after deallocation?
Design choices regarding references and pointers

- Some languages (notably C/C++) distinguish between type $\tau$ and type $\tau^*$ ("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 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
- Advanced uses of types (see for example Rust) can help with this, but remains an active research topic...
Main approaches to deallocation

- C/C++: explicit deallocation (\texttt{free}) must be done by the programmer.
  - (This is very hard to get right, and causes many bugs.)
- Java, Scala, Haskell use \textit{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