Today In A Single Slide

- Memory management has two problems: freeing things too early and freeing things too late.
- Regions are an abstraction in which related objects are allocated together and freed at once.

Memory Management

- Manual memory deallocation is dangerous
  - Deallocate too late ⇒ memory leaks ⇒ performance problems
  - Deallocate too early ⇒ dangling pointers ⇒ safety problems
- Most type-safe languages disallow manual memory deallocation
  - Because their type systems cannot check absence of dangling pointers
  - Such languages use garbage collection ⇒ lack of control
- Question: Is there an effective type system for memory mgmt that allows deallocation?
  - Current best answer: region-based memory management

Regions

- a.k.a. zones, arenas, ...
- Every object is in exactly one region
- Allocation via a region handle
- Deallocate an entire region simultaneously (cannot free an individual object)
- Supports easy serialization

Region-based Memory Management Example

```c
Region r = newregion();
for (i = 0; i < 10; i++) {
    int *x = ralloc(r, (i + 1) * sizeof(int));
    work(i, x);
}
deleteregion(r);
```

Region Expressiveness

- Adds structure to memory management
- Allocate objects into regions based on lifetime
- Works well for objects with related lifetimes
  - e.g., global/per-request/per-phase objects in a server
- Few regions:
  - Easier to keep track of and reason about
  - Delay freeing to convenient “group” time
    - End of an iteration, closing a device, etc
- No writing "free data structure X" functions
Region Expressiveness: lcc
- The lcc C compiler, written using unsafe regions
  - regions bring structure to an application's memory

func
stmt
perm

Time
Safe Region-Based Memory Management

- When is it safe to deallocate a region?
  - Unsafe if you later use a pointer to an object in it!
  - Safe if objects in the same region point to each other
  - But we must handle pointers between regions
- One idea: nested regions lifetimes
  - Use a stack of regions
    - Last region created is also first region deleted
  - Stack frames are a special case of such regions
  - Cannot point from older regions into newer ones
  - Too restrictive in practice
- Today: use a type system to keep track of regions

Region-Flow Type System

- In F₁, we did not model where results of expressions are allocated (e.g., pairs)
  - Now we’ll extend F₁ to track regions
- Specify in what region to store expression results
  - Expr: \( e ::= \lambda x.e \mid e_1 e_2 \mid \ldots \mid e @ \rho \mid e ! \rho \)
  - Region names: \( \rho \) ("rho", Greek letter "r")
- New expressions:
  - \( e @ \rho \) evaluates \( e \) and puts the result in region \( \rho \)
  - We assume that each value lives in a region
  - Think of \( e ! \rho \) as an assertion that value of \( e \) is in region \( \rho \) plus "memcpy e from \( \rho \)"

Example

```plaintext
let cons = \lambda x\lambda y. (x, y) @ \rho_L in
let lst = cons (2 @ \rho_E) (cons (1 @ \rho_E) (0 @ \rho_L)) in
... (fst (lst @ \rho_L)) ! \rho_E ...
```

- Can deallocate \( \rho_L \) without creating dangling pointers
- But if we deallocate \( \rho_E \) first we create dangling pointers

Operational Semantics

- Values live in regions
  - \( v ::= \ldots \mid <v>\rho \)
    - \( <v>\rho \) means value \( v \) living in region \( \rho \)
  - Evaluation rules
    - \( e \rightarrow v \)
    - \( e @ \rho \rightarrow <v>\rho \)
    - \( e ! \rho \rightarrow v \)
  - Evaluation gets stuck if region check ! fails
    - Check: same \( \rho \) above and below line

Typing Rules

- Add a new type to keep track of regions for values
  - \( \tau ::= b \mid \tau_1 \tau_2 \mid \tau @ \rho \)
- Typing rules are straightforward
  - \( \tau \) types keep track of regions of values
  - All values that can flow into one variable must be from the same region
- Soundness result:
  - In well-typed programs the annotations in "\( e ! \rho \)" are always correct
  - i.e., "\( e ! \rho \)" never gets stuck (and can be removed)

Region-Flow Inference

- We start with unannotated programs
- We want to infer the region annotations as follows:
  - Each value constructor \( v \) must be annotated
  - Each deconstructor must be annotated
- \( v ::= n @ \rho \mid (\lambda x.e) @ \rho \)
  - \( e ::= v \mid (e_1 ! \rho) e_2 \mid (fst ! \rho) e \mid if (e ! \rho) then e_1 else e_2 \)
  - We must know, at each use of a value, in what region that value is allocated
### Annotation Example

- We abbreviate:
  
  \[
  n \rho \rightarrow n^\rho \\
  (\lambda x. e) \rho \rightarrow \lambda^\rho x.e \\
  (e_1 ! \rho) e_2 \rightarrow e_1^\rho e_2
  \]

- Consider the code:

  ```
  let fst = \u. \v. u in 
  (let x = \p.(p 0 1) in 
  \q. (q (x fst)) 2 1) 1
  ```

### Region-Flow Type Inference

- Type inference is always possible in this system
- There are multiple correct solutions
  - e.g., use only one region throughout
- There is a "best" solution (up to renaming of regions; best = uses largest # of regions)
  - All other solutions can be obtained by merging some regions in the best solution
- This program analysis is called value-flow analysis
  - Can tell you what values could possibly flow to a use
  - It is a weak form of analysis (equational)
    - For "x := y; x := z;" we get flow between x, y, z (in both directions)

### Adding Region Allocation and Deallocation

- So far we can track (statically) which values are in which region
- We can think of "e @ \rho" as evaluating e and allocating in region \rho space for the result
- We can think of "e ! \rho" as checking that the result of e is in region \rho, and retrieving the result if so
  - The type system tells us that the check is not necessary at run-time. We do not even need to be able to tell at runtime in which region an object is. No tags.
- Still need to know when it is safe to delete a region

### Region Irrelevance

- Assume \( \Gamma \vdash e : \tau \) such that
  - Region \rho is used in \( e \)
  - Region \rho does not appear in \( \Gamma \)
    - Means that before we start \( e \) region \rho is empty
    - Region \rho does not appear in \( \tau \)
      - Means that the result of \( e \) does not refer to any values in \rho
    - The region \rho is relevant only during the execution of \( e \)
- Example:
  - After evaluation of \((\lambda^\rho 0 x. x)^{\rho 0 1} \rho 1\) we can erase \rho 0 if nothing in the context uses it
- Idea: tie region lifetime (relevance) to static scoping

### Statically-Scoped Regions

- Add a new construct
  
  \[
  e ::= \ldots | letreg \rho e
  \]
  - Creates a new region and binds it to the name \rho
  - After \( e \) terminates the region is deleted

  \[
  \Gamma, (R, \rho) \vdash e : \tau \quad \rho \notin RegionVars(\Gamma, \tau)
  \]
  \[
  \Gamma, R \vdash letreg \rho e : \tau
  \]
  \[
  \Gamma, R \vdash e : \tau \quad \rho \in R
  \]

- Example:
  - letreg \rho 0 in (\lambda x. x)^{\rho 0} \tau^\rho is ill typed
  - letreg \rho 0 in (\cons 1^{\rho})^{(\lambda x. x)^{\rho 0} 2^{\rho 0}} is ill typed
  - Type system can detect dangling references. What are they here?

### Unsoundness

- This system works well in first-order languages, where the type of a value fully describes its dependencies
  - A value of type \((\int \rho_1 \times \bool \rho_2) \rho_3\) has references into regions \rho_1 and \rho_2 only
  - A value of type \((\int \rho_1 \times \bool \rho_2) \rho_3\) has references into regions \rho_1 and \rho_2, or \rho_3. Conservatively in \rho_2, \rho_3, and \rho_3
- In higher-order languages we cannot tell so easily
  - letreg \rho 0 in let x = true @ \rho_1 in
    - body of letreg has type \bool@\rho_1 \Rightarrow \bool@\rho_1
      - Later, when \( t \) is used, it will access a dangling pointer to x
- Problem: The type of a function describes only the input/output behavior of the function
  - It does not describe the execution of the function!
Types and Effects

- We enrich the type system to contain information about the computation not just the result.
  - For each computation we keep a set of effects (interesting events that occur as it executes).
- New Judgment: \( \Gamma \vdash e : \phi \tau \)
  - \( e \) computes a value of type \( \tau \) and has effects among those in the set \( \phi \).
- We extend the function types as well:
  - \( \tau ::= \text{int} | \tau @ \rho | \tau_1 \rightarrow \phi \tau_2 \)

Example:
\[
\Gamma \vdash e : \phi \rightarrow \phi_2 \text{int}
\]
- Expression \( e \) evaluates (with effects \( \phi_1 \)) to a function, which when given an \( \text{int} \) evaluates (with effects \( \phi_2 \)) to an \( \text{int} \).

Effects for Regions

- To detect dangling references we need to compute for each expression what set of regions it references at runtime:
  \[
  \Gamma(x) = \tau
  \]
- \( \Gamma \vdash e : \phi \rightarrow \phi_2 \text{int} \)
- \( \Gamma \vdash e : \phi_2 \rightarrow \phi \)
- \( \Gamma \vdash e : \phi_2 \rightarrow \phi \) for \( \rho \notin \text{RegionVars}(\Gamma, \tau) \)

Handling That Old Example

- Consider again the example:
  \[
t = \text{letreg} \rho_0 \text{ in } \text{let } x = \text{true} @ \rho_0 \text{ in } (\lambda y. (\lambda x. x)) z
  \]
- body of letreg has type \( \text{bool} @ \rho_0 \rightarrow \{\rho_0, \rho_1\} \text{bool} @ \rho_1 \)
- Now the type says that \( \rho_0 \) is referenced by the result of \( t \). This program is now ill-typed (i.e., we will notice the region leak).

Effect Types Systems

- We have collected a set of regions referenced.
- Effects can model other intrinsic properties of functions (depending on how the computation proceeds, not only on the result):
  - Behavioral effects
  - Effects now have structure, with sequencing, choice, recursion
- Effects have also been used to model:
  - Cryptographic protocols
  - Synchronization protocols
  - Interference analysis for threads
  - Cleanup actions (previous lecture included a type-and-effect system for compensation stacks)

Soundness

- Here is one way to argue soundness:
  - Soundness = no dangling pointers
- Change the operational semantics of letreg to get stuck if the region is referenced in the result of the body:
  \[
  \rho' = \text{newregion}() \vdash [\rho'/\rho] e \updownarrow_v \rho' \notin \text{RegionVars}(\rho)
  \]
- Prove that well-typed programs never get stuck
- Will this work? Why?

Soundness Problems

- Consider the program:
  \[
t = \text{let } z = 0 @ \rho_0 \text{ in } (\lambda x. (\lambda y. x)) z
  \]
- Type is \( \emptyset \vdash t : \{\rho_0\} \text{int} \rightarrow \emptyset \text{int} \)
- Evaluates to \( t \)'s value = \( \lambda x. (\lambda y. x) \cdot 0 \)
- Not true that RegionVars(\( t \)'s value) = \( \emptyset \)
- Our system does allow dangling pointers
  - But only when you will never dereference them
- In this respect it is more powerful than a garbage collector (able to leap David Bacon in a single bound)
  - Because it can see the rest of the computation
  - The GC only sees a snapshot of the computation state
Soundness Attempt 2

- Introduce a special region called “dangling”
- Replace all dangling regions with this one
- And check that we never use it

\[ \rho' = \text{newregion()} \]
\[ \vdash \rho' \!] e \Downarrow \nu \]
\[ \vdash \text{letreg } \rho \Downarrow \nu \[ \text{[dangling/} \rho \![] ]v \]
\[ \sigma \vdash e \Downarrow \nu \]
\[ \rho \neq \text{dangling} \]
\[ \vdash e ! \rho \Downarrow \nu \]
\[ \sigma \vdash e \Downarrow \nu \]
\[ \rho \neq \text{dangling} \]
\[ \vdash e \Downarrow \nu \]

Prove now that well-typed programs do not get stuck
- No need to introduce the dangling checks at run-time

Region Polymorphism

- Consider this code again
  \[ \text{let cons = } \lambda x. \lambda y. (x, y) \Downarrow \rho L \]
- We need a different function to allocate pairs in different regions. Inconvenient!
- Idea: allow functions to take regions as parameters
- This is called region polymorphism
- We write \[ \text{let cons = } \lambda \rho. \lambda x. \lambda y. (x, y) \Downarrow \rho \]
- Type of result of cons depends on the region argument

\[ \text{Type of cons is } \Pi \rho. \tau_1 \rightarrow \tau_2 \rightarrow (\tau_1 \times \tau_2) \Downarrow \rho \]

Region Polymorphism

- We add the following to the language

\[ e ::= \ldots \mid \lambda \rho. e \] (region abstraction)
\[ | e \rho \] (region application)
\[ \tau ::= \ldots \mid \Pi \rho. \tau \] (region polymorphism)
- In the type \[ \Pi \rho. \tau \] region variable \( \rho \) is bound in \( \phi \) and \( \tau \)

\[ \Gamma \vdash e : \tau \]
\[ \Gamma \vdash e \rho : \Pi \rho. \tau \]
\[ \Gamma \vdash e : \phi' \Pi \rho. \tau \]
- Note that region application does not “reference” the region (it’s purely syntactic, as in “id \[ \text{int} \] 5”)
- More opportunities for harmless dangling references

Effect Polymorphism

- Region polymorphism fails on higher-order languages
- Consider the map function for lists of integers
- Without regions:
  \[ \text{map : (int } \rightarrow \text{int) } \times \text{intlist } \rightarrow \text{intlist} \]
- With regions (potentially moving the list also):
  \[ \text{map : } \Pi \rho. \Pi \rho'. \emptyset (\text{int } \rightarrow \text{int}) \times (\text{intlist } @ \rho) \rightarrow \emptyset (\text{intlist } @ \rho') \]
- But the effect \( \phi \) is hardcoded
- Need a different map for each effect
- \text{Déstá vu: Need effect polymorphism}

Effect Polymorphism

- We do not add syntax for effect polymorphism
  - It is implicit; our type system tracks it
- We add types and typing rules
  \[ e \in \text{EffectVariables} \]
  \[ \tau ::= \ldots \mid \forall \epsilon. \tau \] (very similar to value polymorphism)
- We can now write the \text{map} function:
  \[ \text{map : } \forall \epsilon. \Pi \rho. \Pi \rho'. \emptyset (\text{int } \rightarrow \text{int}) \times (\text{intlist } @ \rho) \rightarrow \emptyset (\text{intlist } @ \rho') \]

Regions In Practice

- Despite heavy use of regions in practice (systems code)
  - Apache, Linux kernel, BerkeleyDB, etc.
- The (formal) study of regions is less than 15 years old
- Few languages include regions
  - MLKit (an implementation of ML)
  - RC (Gay and Aiken, Berkeley)
  - Cyclone (safe variant of C)
- Global region, stack regions, lexically-scoped regions
- All of which failed to set the world on fire ...
- Compromise between complexity of the typing annotations and expressiveness
  - Danger is that the type system may require regions to be long-lived
Homework

- **Project Due**
  - You have less than a fortnight to complete it.
  - Need help? Stop by my office or send email.