Cunning Plan

- Introduction to Regions
- Static and Dynamic Semantics
- Types and Effects
- Safety and Soundness
- Polymorphism

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 mem 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

Region r = newregion();
for (i = 0; i < 10; i++) {
    int *x = malloc(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 need to write “free this data structure” functions
Region Expressiveness: lcc

- The lcc C compiler, written using unsafe regions
  - regions bring structure to an application's memory

```
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

- 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

- Idea: 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 \( \rho \) or “copy \( e \) from \( \rho \)”

Example

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
- If we deallocate \( \rho_E \) first, we create dangling pointers

Operational Semantics

- Values live in regions

\[ v ::= \ldots \mid \langle v \rangle _\rho \]
  - \( \langle v \rangle _\rho \) means value \( v \) living in region \( \rho \)
- Evaluation rules

\[
\begin{align*}
  e @ \rho & \rightarrow \langle v \rangle _\rho \\
  e ! \rho & \rightarrow v
\end{align*}
\]

- Evaluation gets stuck if region check fails

Typing Rules

- Add a new type to keep track of regions for values

\[ \tau ::= b \mid \tau_1 \to \tau_2 \mid \tau @ \rho \]

- Typing rules are straightforward

\[
\begin{align*}
  \Gamma & \vdash e : \tau \\
  \Gamma & \vdash e : \tau @ \rho \\
  \Gamma & \vdash e ! : \tau \rightarrow \tau
\end{align*}
\]

- 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 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

\[
\begin{align*}
  v ::= n @ \rho \mid (\lambda x.e) @ \rho \\
  e ::= v \mid (e_1 @ \rho) e_2 \mid (fst ! \rho) e \\
  & \mid if (e ! \rho) then e_1 else e_2
\end{align*}
\]

- We must know, at each use of a value, in what region that value is allocated
Annotation Example

- We abbreviate:
  - $n \cdot r$ as $n^r$
  - $(\lambda x. e) \cdot r$ as $\lambda^x e$
  - $(e_1 \cdot ! r) \cdot e_2$ as $e_1^r \cdot e_2$

- Consider the code:
  
  ```
  let fst = $\lambda^u. \lambda^v. u$ in 
  (let x = $\lambda^p. (p^1 \cdot 0^0) \cdot 1^1$ in 
  $\lambda^q. (q^1 \cdot (x^0 \cdot \text{fst}))^2 \cdot 2^1 \cdot 1^1$) \cdot \text{fst}
  ```

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 \cdot r" as evaluating e and allocating in region r space for the result
- We can think of "! e \cdot r" as checking that the result of e is in region r, 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 $r$ is used in $e$
  - Region $r$ does not appear in $\Gamma$
    - Means that before we start e region $r$ is empty
  - Region $r$ does not appear in $\tau$
    - Means that the result of e does not refer to any values in $r$
  - The region $r$ is relevant only during the execution of $e$

- Example:
  - After evaluation of $(\lambda x. \text{fst})^1$ we can erase $r_0$ if nothing in the context uses it
- Idea: tie region lifetime (relevance) to static scoping

Statically-Scoped Regions

- Add a new construct
  
  ```
  e ::= ... | \text{letreg } r \text{ in } e
  ```
  - Creates a new region and binds it to the name $r$
  - After e terminates the region is deleted

  $\Gamma, (R, r) \vdash e : \tau \quad r \notin \text{RegionVars}(\Gamma, \tau)$

  $\Gamma, R \vdash \text{letreg } r \text{ in } e : \tau$

- Example:
  - letreg $r_0$ in $(\lambda x. x)^1$ is well typed
  - letreg $r_0$ in $(\lambda x. x)^2$ 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 $(\text{int} \cdot r_0 \times \text{bool} \cdot r_0) \cdot r_1$ has references into regions $r_0$, and $r_1$ only
  - A value of type $(\text{int} \cdot r_0 \times \text{bool} \cdot r_0) \cdot r_1$ has references into regions $r_0$ and ($r_0$ or $r_1$). Conservatively in $r_0$, $r_1$ and $r_2$
- In higher-order languages we cannot tell so easily
  - $t = \text{letreg } r_0$ in let $x = \text{true} \cdot r_0$ in $\lambda y. \text{if } x \text{ then } y \text{ else false}$
    - body of letreg has type bool$^1$ -> bool$^1$
    - 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: $\Gamma \vdash e : \tilde{\phi} \tau$
  - expression $e$ computes a value of type $\tau$ and has effects among those in the set $\tilde{\phi}$

- We extend the function types as well
  $\tau ::= \text{int} | \tau @ \rho | \tau_1 \rightarrow \tau_2$

- Example:
  $\Gamma \vdash e : \tilde{\phi}_1 : \text{int} \rightarrow \tilde{\phi}_2 \text{int}$
  - Expression $e$ evaluates (with effects $\tilde{\phi}_1$) to a function, which when given an int evaluates (with effects $\tilde{\phi}_2$) to an int

Effects for Regions

• To detect dangling references we need to compute for each expression what set of regions it references at runtime

- Example:
  $\Gamma \vdash x : \emptyset \tau$
  $\Gamma(x) = \tau$
  $\Gamma \vdash e_1 : \emptyset \tau \rightarrow \emptyset \tau'$
  $\Gamma \vdash e_2 : \emptyset \utilde{\emptyset} \tau$
  $\Gamma \vdash e_1 e_2 : \emptyset \utilde{\emptyset} \tau$
  $\Gamma \vdash e : \emptyset \tau \rightarrow \emptyset \tau$
  $\Gamma \vdash e@\rho : \emptyset \cup \{\rho\} \tau \rightarrow \emptyset \tau$
  $\Gamma \vdash e!\rho : \emptyset \cup \{\rho\} \tau$
  $\Gamma \vdash \text{letreg } \rho \text{ in } e : \emptyset \cup \{\rho\} \tau$

Handling That Old Example

• Consider again the example
  
  $t = \text{letreg } \rho_0 \text{ in } \lambda x. (\lambda y. x)\ z$

- Type is $\emptyset \vdash t : \\{\rho_0\}\ \text{int} \rightarrow \emptyset\ \text{int}$
- Evaluates to $t$’s value = $\lambda x. (\lambda y.x) <0> \\rho_0$
- Not true that $\text{RegionVars}(t$’s value) = $\emptyset$

- Our system does allow dangling pointers
  - But only when you will never dereference them

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
  $\rho' = \text{newregion}() \vdash [p'/\rho]e \downarrow v \quad p' \notin \text{RegionVars}(v)$
  - Let letreg $\rho$ in $e \downarrow v$
  - Prove that well-typed programs never get stuck
  - Will this work? Why?
Soundness Attempt 2
- Introduce a special region called “dangling”
  - Replace all dangling regions with this one
  - And check that we never use it
- 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
  \[
  \text{let cons} = \lambda \rho. \lambda x. \lambda y. (x, y) @ \rho
  \]
- 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 \( \text{let cons} = \lambda \rho. \lambda x. \lambda y. (x, y) @ \rho \)
  - Type of \text{cons} depends on the region argument
  - Type of \text{cons} is \( \Pi_\rho \tau_1 \to \tau_2 \to (\tau_1 \times \tau_2) @ \rho \)

Effect Polymorphism
- We add the following to the language
  \[
  e ::= ... | \lambda \rho. e \quad (\text{region abstraction}) \\
  | e \rho \quad (\text{region application}) \\
  \tau ::= ... | \Pi_\rho, \tau \quad (\text{region polymorphism})
  \]
- In the type \( \Pi_\rho, \tau \) region variable \( \rho \) is bound in \( \phi \) and \( \tau \)
- Note that region application does not “reference” the region (it’s purely syntactic, as in “id [int] 5”)
- More opportunities for harmless dangling references

Region Polymorphism
- We add types and typing rules
  \[
  \epsilon \in \text{EffectVariables} \\
  \tau ::= ... | \forall \epsilon. \tau \quad (\text{value polymorphism})
  \]
- Very similar to value polymorphism
  \[
  \Gamma \vdash e : \tau \quad \epsilon \notin \text{EffectVars}(\Gamma, \phi) \\
  \Gamma \vdash e : \forall \epsilon. \tau \\
  \Gamma \vdash e : [\phi']\tau \\
  \]
- We can now write the map function:
  \[
  \text{map} : \forall \epsilon. \Pi_\rho. \Pi_\rho', \emptyset \to (\text{intlist} \to \text{intlist}) \\
  \to (\text{intlist} @ p) \to (\text{intlist} @ p')
  \]
  - But the effect \( \phi \) is hardcoded
  - Need a different map for each effect
  - Déjà vu: Need effect polymorphism

Regions In Practice
- Despite heavy use of regions in practice (systems code)
  - The (formal) study of regions is less than 15 years old
  - Few languages include regions
    - MLKit (an implementation of ML)
      - Regions are inferred and used as an implementation mechanism
    - RC (Gay and Aiken, Berkeley)
      - Reference counting of inter-region pointers
    - Cyclone (safe variant of C)
      - Somewhat lighter-weight
      - 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 Tue Apr 25
  - You have **FIVE DAYS** to complete it.
  - Need help? Stop by my office or send email.