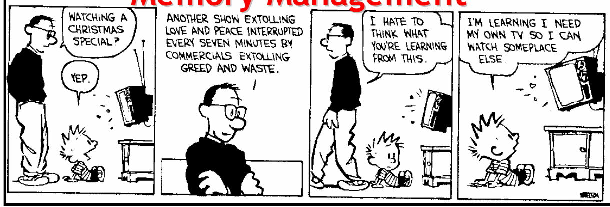

# Region-Based Memory Management




## Cunning Plan

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

#2

## Memory Management

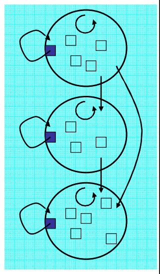
- Manual memory deallocation is dangerous
  - Deallocate too late ⇒ memory leaks ⇒ performance problems
  - Deallocate too early ⇒ dangling pointers ⇒ safety problems
- Most type-safe languages disallow manual memory deallocation
  - Because their type systems cannot check absence of dangling pointers
  - 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

#3

## 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



#4

## Region-based Memory Management Example

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

#5

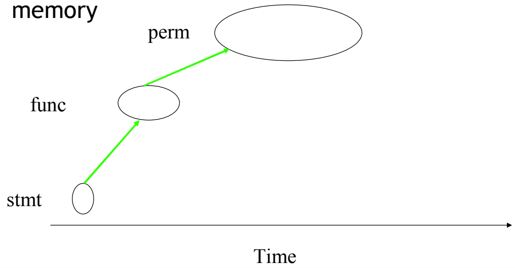
## Region Expressiveness

- 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

#6

# Region Expressiveness: lcc

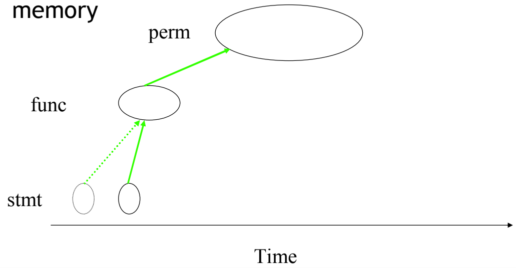
- The lcc C compiler, written using unsafe regions
  - regions bring structure to an application's memory




#7

# Region Expressiveness: lcc

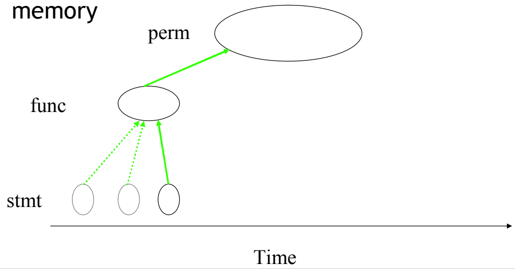
- The lcc C compiler, written using unsafe regions
  - regions bring structure to an application's memory




#8

# Region Expressiveness: lcc

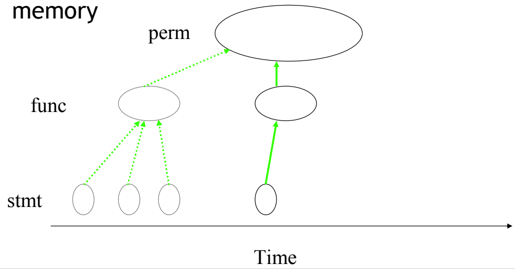
- The lcc C compiler, written using unsafe regions
  - regions bring structure to an application's memory




#9

# Region Expressiveness: lcc

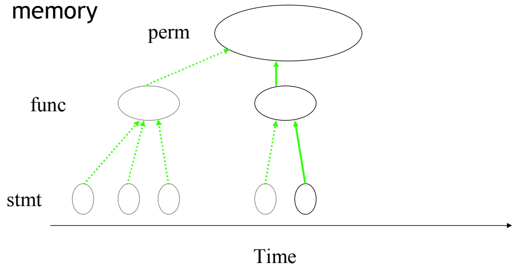
- The lcc C compiler, written using unsafe regions
  - regions bring structure to an application's memory




#10

# Region Expressiveness: lcc

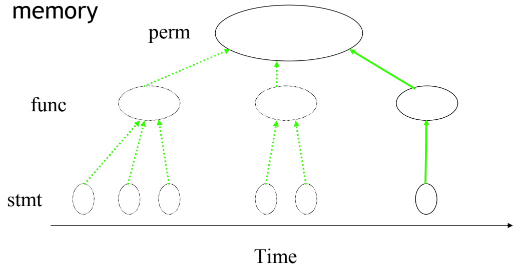
- The lcc C compiler, written using unsafe regions
  - regions bring structure to an application's memory




#11

# Region Expressiveness: lcc

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




#12

## Safe Region-Based Memory Management

- When is it *safe to deallocate* a region?
  - Unsafe if you later user 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 $F_1$ we *did not model* where results of expressions are allocated (e.g., pairs)
  - Now we'll extend $F_1$ to track regions
- Specify in what region to store expression results

Expr: $e ::= \lambda x.e \mid e_1\ e_2 \mid \ldots \mid e\ @\ \rho \mid e\ !\ \rho$

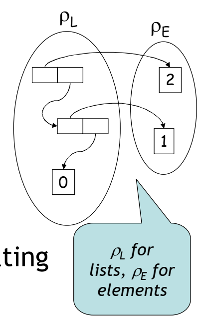
Region names: $\rho$ ("rho", Greek letter "r")

- 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 $\rho_L$ witout 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

$$\frac{e \to v}{e\ @\ \rho \to \langle v\rangle_\rho} \qquad \frac{e \to \langle v\rangle_\rho}{e\ !\ \rho \to v}$$

- 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

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e\ @\ \rho : \tau\ @\ \rho} \qquad \frac{\Gamma \vdash e : \tau\ @\ \rho}{\Gamma \vdash e\ !\ \rho : \tau}$$

- 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 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

$v ::= n\ @\ \rho \mid (\lambda x.e)\ @\ \rho$

$e ::= v \mid (e_1\ !\ \rho)\ e_2 \mid (\text{fst}\ !\ \rho)\ e$

$\mid \text{if}\ (e\ !\ \rho)\ \text{then}\ e_1\ \text{else}\ e_2$

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

## Annotation Example

- We abbreviate:

| | | |
|---|---|---|
| $n @ \rho$ | as | $n^{\rho}$ |
| $(\lambda\ x.\ e) @ \rho$ | as | $\lambda^{\rho}\ x.e$ |
| $(e_1\ !\ \rho)\ e_2$ | as | $e_1{}^{\rho}\ e_2$ |

- Consider the code:

$$\text{let fst} = \lambda^{\rho_f} u.\ \lambda^{\rho_a} v.\ u \text{ in}$$

$$(\text{let } x = \lambda^{\rho_x} p.(p^{\rho_f}\ 0^{\rho_0})^{\rho_a}\ 1^{\rho_1}$$

$$\text{in } \lambda^{\rho_q} q.\ (q^{\rho_f}\ (x^{\rho_x} \text{fst}))^{\rho_a}\ 2^{\rho_1})^{\rho_q}\ \text{fst}$$

#19

## 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)

#20

## 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

#21

## 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

#22

## Statically-Scoped Regions

- Add a new construct

$$e ::= \ldots \mid \text{letreg } \rho \text{ in } e$$

  - Creates a new region and binds it to the name $\rho$
  - After e terminates the region is deleted

$$\frac{\Gamma, (R, \rho) \vdash e : \tau \qquad \rho \notin \text{RegionVars}(\Gamma, \tau)}{\Gamma, R \vdash \text{letreg } \rho \text{ in } e : \tau}$$

$$\frac{\Gamma, R \vdash e : \tau @ \rho}{\Gamma, R \vdash e\ !\ \rho : \tau} \qquad \frac{\Gamma, R \vdash e : \tau \qquad \rho \in R}{\Gamma, R \vdash e @ \rho : \tau @ \rho}$$

- Example:
  letreg $\rho_0$ in $(\lambda^{\rho_0} x.\ x)^{\rho_0}\ 1^{\rho_1}$ is well typed
  letreg $\rho_0$ in (cons $1^{\rho_1}$ $((\lambda^{\rho_0} x.\ x)^{\rho_0}\ 2^{\rho_0}))^{\rho_1}$ is ill typed
    - Type system can detect dangling references. What are they here?

#23

## Unsoundness

- This system works well in first-order languages, where the type of a value fully describes its dependencies
  - A value of type $(\text{int} @ \rho_1 \times \text{bool} @ \rho_2) @ \rho_1$ has references into regions $\rho_1$ and $\rho_2$ only
  - A value of type $(\text{int} @ \rho_1 + \text{bool} @ \rho_2) @ \rho_3$ has references into regions $\rho_3$ and ($\rho_1$ or $\rho_2$). Conservatively in $\rho_1$, $\rho_2$ and $\rho_3$
- In higher-order languages we cannot tell so easily
  t = letreg $\rho_0$ in let x = true @ $\rho_0$ in
  $\lambda$ y.if x ! $\rho_0$ then y else false @ $\rho_1$
  - body of letreg has type $\text{bool}^{\rho_1} \to \text{bool}^{\rho_1}$
  - 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!

#24

## 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: $\Gamma \vdash e :^{\phi} \tau$
  - expression e computes a value of type $\tau$ and has effects among those in the set $\phi$
- We extend the function types as well

$$\tau ::= \text{int} \mid \tau @ \rho \mid \tau_1 \to^{\phi} \tau_2$$

- Example:

$$\Gamma \vdash e :^{\phi 1} : \text{int} \to^{\phi 2} \text{int}$$

  - Expression e evaluates (with effects $\phi_1$) to a function, which when given an int evaluates (with effects $\phi_2$) to an int

#25

## Effects for Regions

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

$$\frac{\Gamma, x : \tau \vdash e :^{\phi} \tau}{\Gamma \vdash \lambda x.\ e :^{\emptyset} \tau_1 \to^{\phi} \tau_2}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x :^{\emptyset} \tau} \qquad \frac{\Gamma \vdash e_1 :^{\phi 1} \tau \to^{\phi} \tau' \qquad \Gamma \vdash e_2 :^{\phi 2} \tau}{\Gamma \vdash e_1\ e_2 \ :^{\phi 1 \cup \phi 2 \cup \phi}\ \tau'}$$

$$\frac{\Gamma \vdash e :^{\phi} \tau}{\Gamma \vdash e @ \rho :^{\phi \cup \{\rho\}} \tau @ \rho} \qquad \frac{\Gamma \vdash e :^{\phi} \tau @ \rho}{\Gamma \vdash e\,!\,\rho :^{\phi \cup \{\rho\}} \tau}$$

$$\frac{\Gamma \vdash e :^{\phi} \tau \qquad \rho \notin \text{RegionVars}(\Gamma, \tau)}{\Gamma \vdash \text{letreg } \rho \text{ in } e \quad :^{\phi - \{\rho\}} \quad \tau}$$

#26

## Handling That Old Example

- Consider again the example

  t = letreg $\rho_0$ in

  let x = true @ $\rho_0$ in

  $\lambda$ y. if x ! $\rho_0$ then y else false @ $\rho_1$

  - body of letreg has type

$$\text{bool} @ \rho_1 \to^{\{\rho 0,\ \rho 1\}} \text{bool} @ \rho_1$$

- 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).

#27

## 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)

#28

## 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

$$\frac{\rho' = \text{newregion}() \qquad \vdash [\rho'/\rho]e \Downarrow v \qquad \rho' \notin \text{RegionVars}(v)}{\vdash \text{letreg } \rho \text{ in } e \Downarrow v}$$

- Prove that well-typed programs never get stuck
- Will this work? Why?

#29

## Soundness Problems

- Consider the program

$$t = \text{let } z = 0 @ \rho_0 \text{ in } \lambda\ x.(\lambda\ y.\ x)\ z$$

  - Type is $\emptyset \vdash t : ^{\{\rho 0\}} \text{int} \to^{\emptyset} \text{int}$
  - Evaluates to t's value = $\lambda$ x.($\lambda$ y.x) $\langle 0 \rangle_{\rho_0}$
  - 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

#30

## Soundness Attempt 2

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

$$\frac{\rho' = \text{newregion}() \qquad \vdash [\rho'/\rho]e \Downarrow v}{\vdash \text{letreg } \rho \text{ in } e \Downarrow [\text{dangling}/\rho']v}$$

$$\frac{\sigma \vdash e \Downarrow \langle v \rangle_\rho \qquad \rho \neq \text{dangling}}{\vdash e\ !\ \rho \Downarrow v}$$

$$\frac{\sigma \vdash e \Downarrow v \qquad \rho \neq \text{dangling}}{\vdash e\ @\ \rho \Downarrow \langle v \rangle_\rho}$$

- 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
  let cons = $\lambda$ x$\lambda$ y. (x, y) @ $\rho_L$
- 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 let cons = $\lambda\rho.\ \lambda$ x. $\lambda$ y. (x, y) @ $\rho$
- Type of result of cons depends on the region argument
- Type of cons is $\Pi\rho.\tau_1 \to \tau_2 \to (\tau_1 \times \tau_2)\ @\ \rho$



## Region Polymorphism

- We add the following to the language

| | |
|---|---|
| $e ::= \dots \mid \lambda\ \rho.\ e$ | (region abstraction) |
| $\mid e\ \rho$ | (region application) |
| $\tau\ ::= \dots \mid \Pi\rho.^{\phi}\ \tau$ | (region polymorphism) |

  - In the type $\Pi\rho.^{\phi}\ \tau$ region variable $\rho$ is bound in $\phi$ and $\tau$

$$\frac{\Gamma \vdash e :^{\phi} \tau}{\Gamma \vdash \lambda\rho.\ e :^{\emptyset} \Pi\rho.^{\phi}\ \tau} \qquad \frac{\Gamma \vdash e :^{\phi'} \Pi\rho.^{\phi}\ \tau}{\Gamma \vdash e\ \rho'\ :^{\phi' \cup [\rho'/\rho]\phi}\ [\rho'/\rho]\tau}$$

  - Note that region application does not "reference" the region (it's purely syntactic, as in "`id [int] 5`")
  - More opportunities for harmless dangling references



## Effect Polymorphism

- Region polymorphism fails on higher-order languages
- Consider the map function for lists of integers
- Without regions:
  map : (int $\to$ int) $\times$ intlist $\to$ intlist
- With regions (potentially moving the list also):
  map : $\Pi\rho.\Pi\rho'.^{\emptyset}$ (int $\to^{\phi}$ int) $\times$ (intlist @ $\rho$) $\to^{\phi \cup \{\rho,\rho'\}}$ (intlist @ $\rho'$)
  - But the effect $\phi$ is hardcoded
  - Need a different map for each effect
- Déjà 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

$\varepsilon \in$ EffectVariables

$\tau ::= \dots \mid \forall\varepsilon.\ \tau$

  - Very similar to value polymorphism

$$\frac{\Gamma \vdash e :^{\phi} \tau \qquad \varepsilon \notin \text{EffectVars}(\Gamma, \phi)}{\Gamma \vdash e :^{\phi} \forall\varepsilon.\tau} \qquad \frac{\Gamma \vdash e :^{\phi} \forall\varepsilon.\tau}{\Gamma \vdash e :^{\phi} [\phi'/\varepsilon]\tau}$$

- We can now write the map function:
  map : $\forall\varepsilon.\Pi\rho.\Pi\rho'.^{\emptyset}$ (int $\to^{\varepsilon}$ int) $\times$ (intlist @ $\rho$) $\to^{\varepsilon \cup \{\rho,\rho'\}}$ (intlist @ $\rho'$)



## 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.

#37