Simplifying Regions

Greg Morrisett

Harvard University

Collaborators: Matthew Fluet & Amal Ahmed

The Cyclone Safe-C Project

Primary goal: type-safety

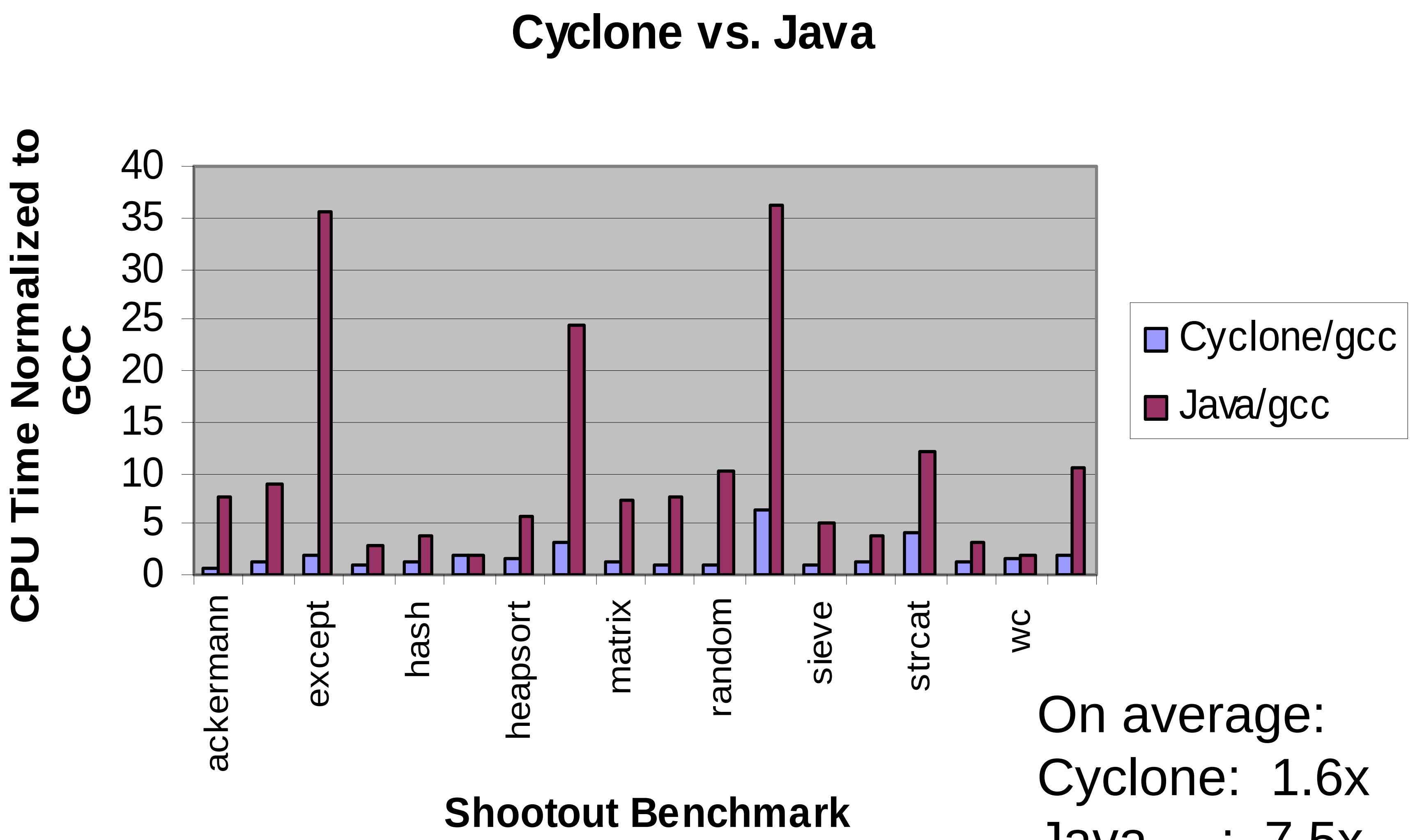
Secondary goal: retain virtues of C

- C programmers should feel comfortable.
- It should be easy to interoperate with legacy C.
- Most importantly, costs should be manifest:
  - Programmers can understand the physical layout of data structures by looking at the types.
  - Programmers can avoid overheads of run-time tags and checks by programming with certain idioms.
  - Want this to be suitable for real-time and embedded settings where space and time may be scarce.

Some Cyclone Users

- In-kernel Network Monitoring [Penn]
- MediaNet [Maryland & Cornell]
- Open Kernel Environment [Leiden]
- RBClick Router [Utah]
- xTCP [Utah & Washington]
- Lego Mindstorm on BrickOS [Utah]
- Cyclone on Nintendo DS [AT&T]
  - Scheme run-time & interpreter
- Cyclone compiler, tools, & libraries
  - Over 100 KLOC
  - Plus many sample apps, benchmarks, etc.

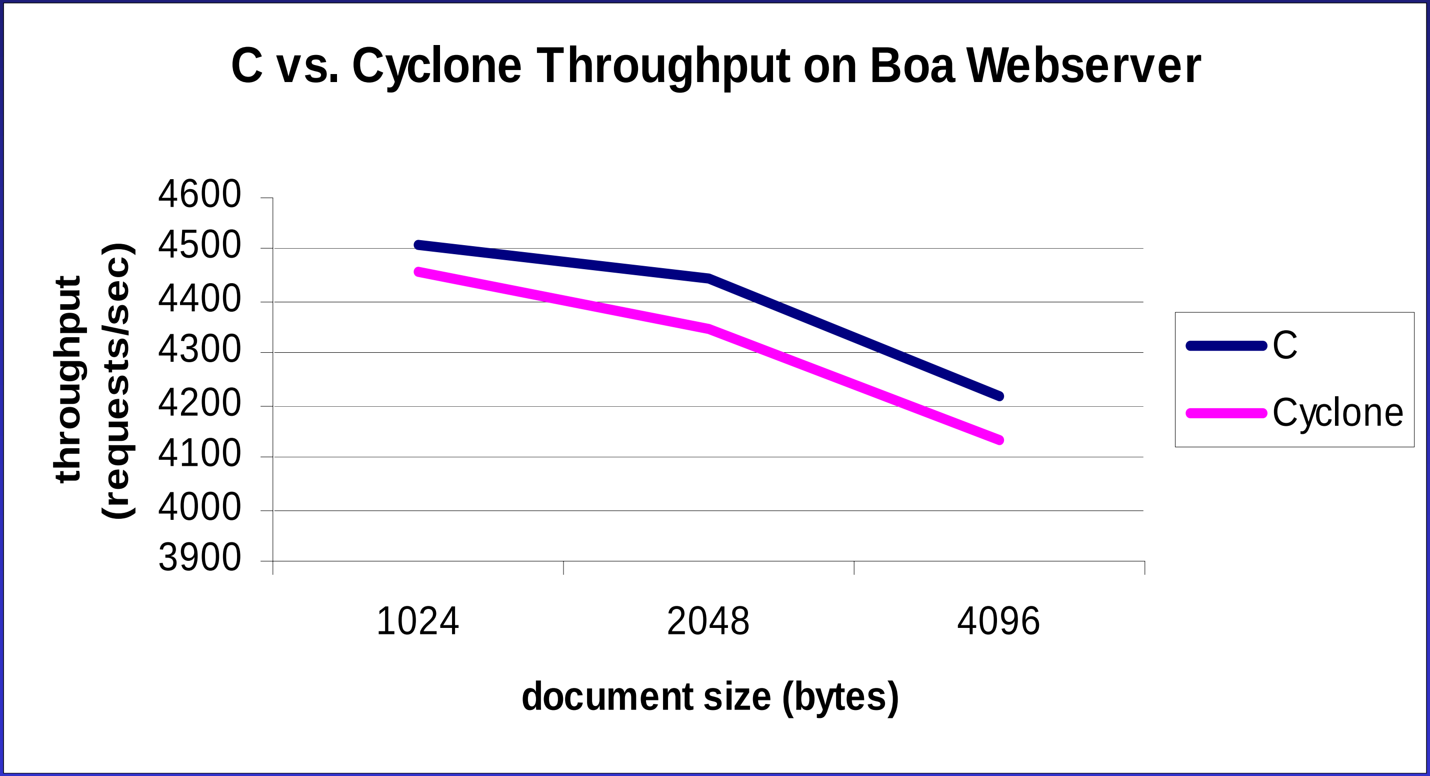
C vs. Cyclone vs. Java



On average:
Cyclone: 1.6x
Java : 7.5x

Macro-benchmarks:

We have also ported a variety of security-critical applications where we see little overhead (e.g., 3% throughput for the Boa Webserver.)



Memory Management

A range of options:

- Heap allocation with conservative GC
- Lexical Regions
  - Stack allocation
  - Lexical arena allocation
  - Tofte & Talpin + region subtyping
- 1st class Regions
  - Enables “tail-calls” -- can code copying GC
- Unique pointers
  - Enables reclamation of individual objects

Each has different tradeoffs.

The Flexibility Pays: MediaNET

TTCP benchmark (packet forwarding):

Cyclone v.0.1 (lexical regions & BDW GC)

- High water mark: 840 KB
- 130 collections
- Basic throughput: 50 MB/s

Cyclone v.0.5 (unique ptrs + dynamic regions)

- High water mark: 8 KB
- 0 collections
- Basic throughput: 74MB/s

A Model?

The combination of lexical regions, unique pointers, region subtyping, etc. makes the meta-theory of Cyclone a nightmare.

- Gave up on usual syntactic proof.

At the heart of the problem:

- Certain types are “ephemeral”.
- The interaction between persistent and ephemeral types is extremely subtle.
- Polymorphism really complicates things.
- Same issue arises in many other settings: TAL(T), Vault, Cqual, Haskell’s runST, …

Outline

Core Cyclone $\rightarrow$ F+RGN [ICFP'04]

- Effects map to an indexed store monad
- Coercion-based interpretation of subtyping

F+RGN $\rightarrow$ Linear F+Stores

- Monad abandoned in favor of linearity.
- Regions become 1st-class, unique pointers fall out as a special case.
- Developing a semantic model of the target.
- Believe it serves as foundation for Cqual, Vault, etc.

The Tofte-Talpin Region Calculus

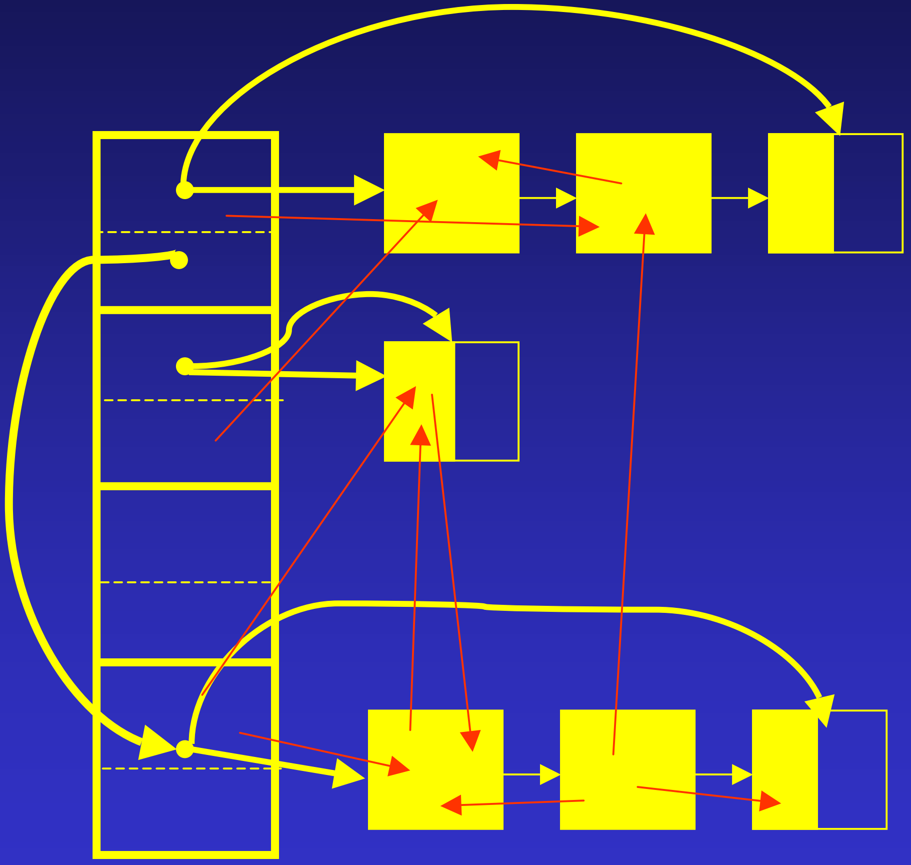
Operationally:

- Memory is divided into *regions* (ρ)
- Objects are allocated in a region: (3,2)@ρ
- Regions are created and destroyed with a lexically-scoped construct:

  `letregion` ρ **`in`** e

- All objects allocated in ρ are deallocated at the end of ρ's scope.
- Region names can be passed into functions to support a "callee-allocates in caller's region idiom."

Runtime Organization



runtime stack

Regions are linked lists of pages.

Arbitrary inter-region references.

Similar to arena-style allocators.

Typing

- Pointer types indicate referent's region: $(\text{int},\text{int})@\rho$
- The type system tracks the set φ of regions that are accessed when a computation is run: $\Gamma \vdash e : T, \varphi$
- Function types include a latent effect:

$$T_1 \xrightarrow{\varphi} T_2$$

- The role of φ is to tell us when it's *not* safe to deallocate a region.

Letregion

The typing for **`letregion`** is subtle:

$$\frac{\Gamma \vdash e : \tau, \varphi \qquad \rho \notin FRV(\Gamma,\tau)}{\Gamma \vdash \texttt{letregion}\ \rho\ \texttt{in}\ e : \tau, \varphi \backslash \rho}$$

In particular, pointers into ρ can escape the scope of the letregion.

Example:

letregion ρ **in**

  let x = (1,2)@ ρ **in**

  let z = (3,4)@ ρ' **in**

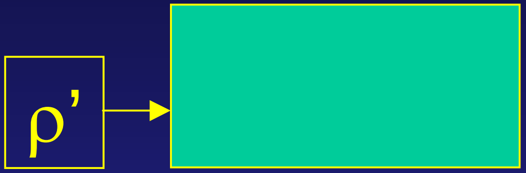
  let w = (x,z)@ ρ' **in**

  $\lambda y.\#1(\#2\ w) + y \quad : \quad \text{int} \xrightarrow{\{\rho'\}} \text{int}, \{\rho'\}$

Example:

```
letregion ρ in
  let x = (1,2)@ ρ in
  let z = (3,4)@ ρ' in
  let w = (x,z)@ ρ' in
  λy.#1(#2 w) + y      :    int →{ρ'} int, {ρ'}
```

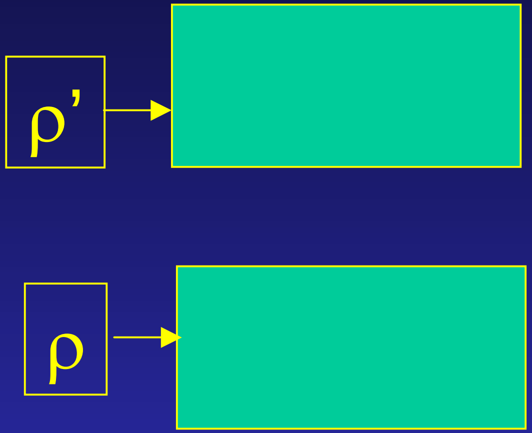
$\lambda y.\#1(\#2\ w) + y \quad : \quad \text{int} \xrightarrow{\{\rho'\}} \text{int}, \{\rho'\}$



Example:

```
letregion ρ in
  let x = (1,2)@ ρ in
  let z = (3,4)@ ρ' in
  let w = (x,z)@ ρ' in
  λy.#1(#2 w) + y     :   int →{ρ'} int, {ρ'}
```

$$\text{int} \xrightarrow{\{\rho'\}} \text{int}, \{\rho'\}$$



Example:

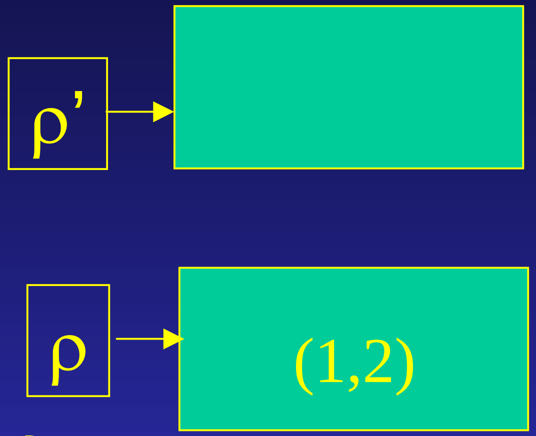
letregion ρ **in**

  let x = (1,2)@ ρ **in**

  let z = (3,4)@ ρ' **in**

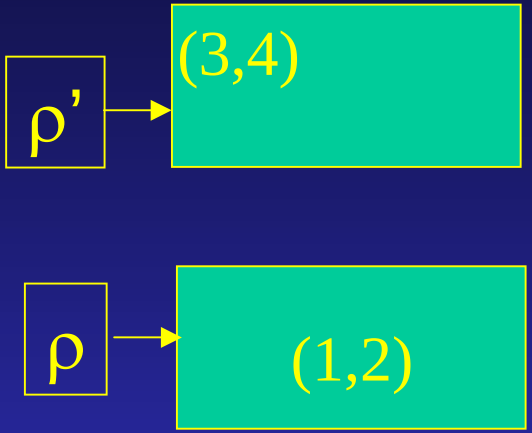
  let w = (x,z)@ ρ' **in**

  λy.#1(#2 w) + y : $\text{int} \xrightarrow{\{\rho'\}} \text{int}, \{\rho'\}$



Example:

letregion ρ **in**
  let x = (1,2)@ ρ **in**
  $\underline{\textbf{let } z = (3,4)@\ \rho' \textbf{ in}}$
  let w = (x,z)@ ρ' **in**
  λy.#1(#2 w) + y : $\text{int} \xrightarrow{\{\rho'\}} \text{int}, \{\rho'\}$



Example:

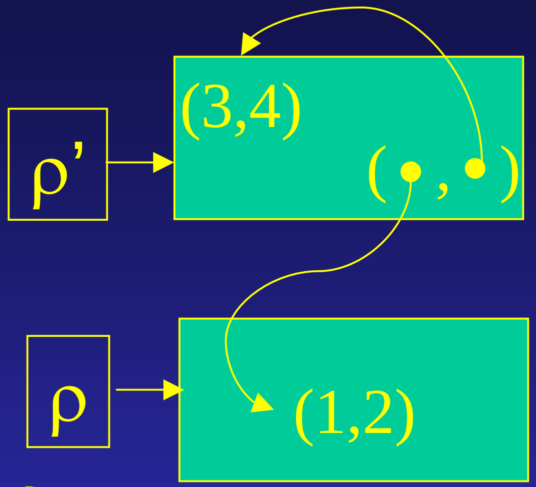
letregion ρ **in**

  let x = (1,2)@ ρ **in**

  let z = (3,4)@ ρ' **in**

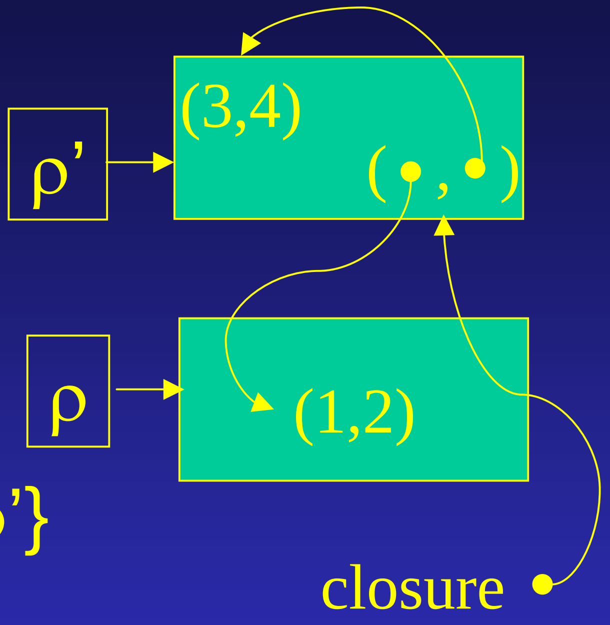
  let w = (x,z)@ ρ' **in**

  λy.#1(#2 w) + y   : $\text{int} \xrightarrow{\{\rho'\}} \text{int}, \{\rho'\}$



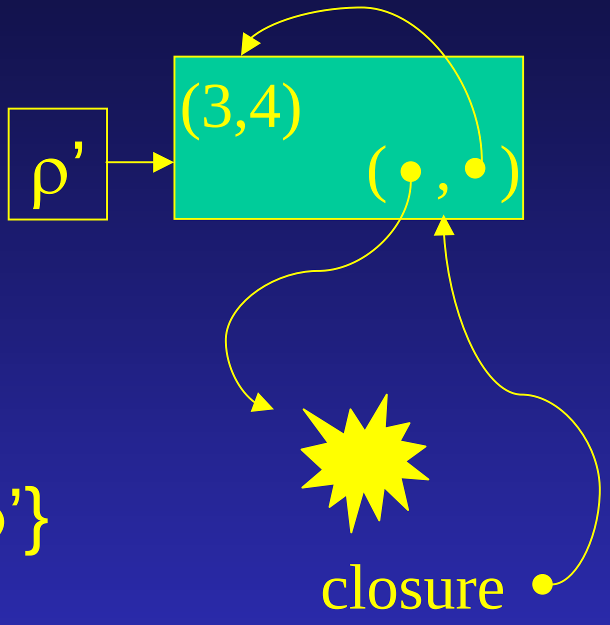
Example:

letregion ρ **in**
  let x = (1,2)@ ρ **in**
  let z = (3,4)@ ρ' **in**
  let w = (x,z)@ ρ' **in**
  <u>λy.#1(#2 w) + y</u>   : $\text{int} \xrightarrow{\{\rho'\}} \text{int}, \{\rho'\}$



Example:

letregion ρ **in**
  let x = (1,2)@ ρ **in**
  let z = (3,4)@ ρ' **in**
  let w = (x,z)@ ρ' **in**
  λy.#1(#2 w) + y   :   int $\xrightarrow{\{\rho'\}}$ int, $\{\rho'\}$



Pointers are persistent, regions aren't…

Subtyping

Tofte & Talpin's effect weakening:

$$\frac{\Gamma \blacktriangleright e : \tau, \varphi \qquad \varphi \subseteq \varphi'}{\Gamma \blacktriangleright e : \tau, \varphi'}$$

Cyclone's region "outlives":

$$\frac{\Gamma \blacktriangleright \rho \leq \rho'}{\Gamma \blacktriangleright \tau@\rho \leq \tau@\rho'}$$

$$\frac{\Gamma, \mathrm{FRV}(\Gamma) \leq \rho \blacktriangleright e : \tau, \varphi \qquad \rho \notin \mathrm{FRV}(\Gamma,\tau)}{\Gamma \blacktriangleright \texttt{letregion}\ \rho\ \texttt{in}\ e : \tau, \varphi\backslash\rho}$$

Core Cyclone to F+RGN

The source language is complicated by:

- Effects: sets of regions
- Subtyping, letregion, polymorphism.

Choose as intermediate language:

- CBV System-F plus…
- An indexed monad family: RGN σ τ
  - Inspired by Haskell’s ST monad.
  - Key: run can be provided in the language.
- Eliminate subtyping via coercions

Type Constructors

RGN $\sigma\ \tau$

computation running in store σ producing a τ.

ptr $\rho\ \tau$

pointer into region ρ holding a τ value.

$\rho \in \sigma$

a proof that σ includes the region ρ

$\sigma_1 \leq \sigma_2$ $\quad$ $[= \mathbf{8}\rho.(\rho \in \sigma_1)\ \mathbf{!}\ (\rho \in \sigma_2)]$

a proof of store inclusion

Translation Essence:

«int@$\rho_1 \overset{\{\rho_1,\rho_2,\rho_3\}}{\overline{}}$! int@$\rho_3$¬ ¼

8 σ. ($\rho_1 \in \sigma$) **!** ($\rho_2 \in \sigma$) **!** ($\rho_3 \in \sigma$) **!**
(ptr ρ_1 int) **!** RGN σ (ptr ρ_3 int)

Monadic Operations

`return` : **8**α,σ. α **!** RGN σ α

`then` : **8**α,β,σ. RGN σ α **!**

(α **!** RGN σ β) **!** RGN σ β

- Can only sequence in *same* store.
- Need some way to lift computations in sub-stores

`run` : **8**α. (**8**σ. RGN σ α) **!** α

- Note that α cannot mention σ!
- Quite similar to letregion.

Primitives:

`new`:

$8\alpha,\sigma,\rho.\ \alpha\ !\ (\rho \in \sigma)\ !\ \mathrm{RGN}\ \sigma\ (\mathrm{ptr}\ \rho\ \alpha)$

`read`:

$8\alpha,\sigma,\rho.\ \mathrm{ptr}\ \rho\ \alpha\ !\ (\rho \in \sigma)\ !\ \mathrm{RGN}\ \sigma\ \alpha$

`letRGN` :

$8\alpha,\sigma_1.\ (8\sigma_2.\ (\sigma_1 \leq \sigma_2)\ !\ (\rho \in \sigma_2)\ !\ \mathrm{RGN}\ \sigma_2\ \alpha)$
$\quad\quad !\ \mathrm{RGN}\ \sigma_1\ \alpha$

`subRGN` :

$8\alpha,\sigma_1,\sigma_2.\ (\sigma_1 \leq \sigma_2)\ !\ \mathrm{RGN}\ \sigma_1\ \alpha\ !\ \mathrm{RGN}\ \sigma_2\ \alpha$

Notes:

We constructed an operational model and proved a soundness result at this level, as well as the correctness of the translation.

In practice, you need to phase-split the evidence (e.g., $\rho \in \sigma$) and coercions.

F+RGN is somewhat simpler than T.T. and sheds light on regions and Haskell's ST, but not 1st class regions or unique pointers.

New Target: Linear F + regions

- We'll use a *linear* version of F similar to Walker & Watkins.
- We'll eliminate the RGN monad in favor of explicit store-passing but use linearity to ensure store remains single-threaded.
- Unique pointers & 1st class regions pop out for free…

Types:

$T ::= \alpha \mid \text{int}$

| ptr ρ T (pointer into region ρ)

| cap ρ (capability for region ρ)

| 1 | $T_1 \otimes T_2$

| $T_1 \multimap T_2$

| !T

| 8α.T | 8ρ.T

| $\exists\alpha$.T | $\exists\rho$.T

Primitives:

newrgn : $1 \multimap \exists\rho.\mathrm{cap}\ \rho$

freergn : $8\rho.\mathrm{cap}\ \rho \multimap 1$

new : $8\alpha,\rho.!\alpha \multimap \mathrm{cap}\ \rho \multimap \mathrm{cap}\ \rho \otimes !\mathrm{ptr}\ \rho\ !\alpha$

read : $8\alpha,\rho.\mathrm{ptr}\ \rho\ !\alpha \multimap \mathrm{cap}\ \rho \multimap \mathrm{cap}\ \rho \otimes !\alpha$

Dynamics

Mostly just CBV lambda calculus.

Semantic values:

- ptr ρ τ $\approx$ Loc$_\rho$
- cap ρ $\approx$ Loc$_\rho \to$ Val
- NB: !(cap ρ) $\approx \emptyset$

We actually use a step-indexed model a la Appel & McAllester to avoid problems with recursive types.

Encoding F+RGN Types

«int¬ = !«int¬

«ptr σ τ¬ = !ptr σ «τ¬

«τ_1 ! τ_2¬ = !(«τ_1¬ —∘ «τ_2¬)

«RGN σ τ¬ = σ —∘ σ ⊗ «τ¬

«ρ ∈ σ¬ = ! ∃σ’. (σ —∘ σ’ ⊗ cap ρ) ⊗
(σ’ ⊗ cap ρ —∘ σ)

«$\sigma_1 \leq \sigma_2$ ¬ = ! ∃σ’. (σ_2 —∘ σ_1 ⊗ σ’) ⊗
(σ_1 ⊗ σ’ —∘ σ_2)

Encoding Monadic Primitives:

Just store-passing:

$\ulcorner$**`return`**$\urcorner = \Lambda\alpha,\sigma.\ \lambda x{:}!\alpha.\ \lambda s{:}\sigma.\ (s,x)$

$\ulcorner$**`then`**$\urcorner = \Lambda\alpha,\beta,\sigma.$

$\lambda f{:}\ulcorner \mathrm{RGN}\ \sigma\ \alpha \urcorner.$

$\lambda g{:}!(!\alpha \multimap \ulcorner \mathrm{RGN}\ \sigma\ \beta \urcorner).$

$\lambda s{:}\sigma.$ **`let`** $(s', y) = f\ s$ **`in`** $g\ y\ s'$

Encoding Let-region

«letRGN¬ =

$\Lambda\alpha,\sigma_1.\lambda$f: **«**8$\sigma_2$. $\sigma_1 \leq \sigma_2$! $\rho \in \sigma_2$! RGN σ_2 α **¬**.

λs:σ_1.

unpack [ρ,c] = newrgn () **in**

let w_2 = pack[σ_1,(id,id)]:**«**$\rho \in (\sigma_1 \otimes$ cap ρ)**¬** **in**

let w_1 = pack[cap ρ,(id,id)]:**«**$\sigma_1 \leq (\sigma_1 \otimes$ cap ρ)**¬**

in **let** ((s,c),x) = f [$\sigma_1 \otimes$ cap ρ] w_1 w_2 (s,c) **in**

freergn c;

(s,x)

Key: new store is $\sigma_1 \otimes$ cap ρ

Encoding New and Read:

Use witnesses to get capability from store:

```
«new¬ = Λα,σ,ρ. λx:!α. λw:«ρ ∈ σ¬.λs:σ.
            unpack [σ’,(f,g)] = w in
            let (s’,c) = f s in
            let (c,r) = new x c in
            let s = g (s’,c) in (s,r)
«read¬ = Λα,σ,ρ.λx:ptr ρ !α. λw:«ρ ∈ σ¬.λs:σ.
            unpack [σ’,(f,g)] = w in
            let (s’,c) = f s in
            let (c,x) = read r c in
            let s = g (s’,c) in (s,r)
```

Subrgn

Use witness to get sub-store:

```
«subRGN¬ =
    $\Lambda\alpha,\sigma_1,\sigma_2.\ \lambda w{:}«\sigma_1 \le \sigma_2¬.\ \lambda k{:}«\text{RGN}\ \sigma_1\ \alpha¬.$
     $\lambda s_2{:}\sigma_2.$
      unpack [$\sigma'$,(f,g)] = w in
      let ($s_1$,s') = f $s_2$ in
      let ($s_1$,x) = k $s_1$ in
      let $s_2$ = g ($s_1$,s') in  ($s_2$,x)
```

1st Class Regions

At the target level, regions are 1st class!

- Can export newrgn & freergn to the source.
- No LIFO constraints needed!
- Source-level 1st class region: $\exists\rho.(\text{cap } \rho \otimes !T[\rho])$

We can *open* such a region to regain the convenience of the monadic threading:

8ρ.cap ρ —◦

8α,σ_1. (8σ_2.«$\sigma_1 \leq \sigma_2$¬ —◦ «$\rho \in \sigma_2$¬ —◦ «RGN σ_2 α¬)

—◦ RGN σ_1 (cap ρ ⊗ α)

- So the monad is purely a convenience.

Unique Pointers

These are just a degenerate case of 1st class regions: $\exists\rho.(\text{cap } \rho \otimes \text{!ptr } \rho\ \tau)$

We can deallocate these at will!

- In practice, we split cap ρ into two capabilities.
- One ($\underline{\text{access } \rho}$) lets us access ρ.
- The other (alloc ρ) lets us allocate in ρ.
- Only the alloc capability is needed at run-time.
- So a unique pointer is: $\exists\rho.(\underline{\text{access } \rho} \otimes \text{!ptr } \rho\ \tau)$
- Can “open” a unique pointer to again regain convenience of monadic abstraction.

Recap:

- At source-level, we seem to have a variety of memory mgmt. facilities:
  - Stack allocation, lexical regions, 1st class regions, unique pointers, …
  - They’re all useful in practice.
- The target exposes the commonalities:
  - Linear capabilities for access control ensure state is single-threaded *and* eventually reclaimed.
  - Monadic encapsulation is purely a convenience (implicit threading of capabilities).
  - That convenience has a price: LIFO.
  - Fortunately, we don’t *have* to encapsulate.

Future Work:

- Need to fill in all of the details.
- Need to phase-split capabilities.
- In practice, need affine, linear, and unrestricted types to model Cyclone.
- Modeling other languages:
  - Alias types, Cqual: require only a slight refinement where we have two kinds of pointers (ephemeral vs. persistent).
  - Vault: still need to account for adoption and suspect that relevant types play role.