Region Type Systems and Inference— Lecture 3/3

Martin Elsman, DIKU

Seminar at Chalmers University of Technology — December 18–19, 2018

- The region-based memory model
- A type-and-effect system for region-based memory management
- Region- and effect-polymorphism
- Region inference and arrow effects
- Combining Region-inference and garbage collection
- Exercises

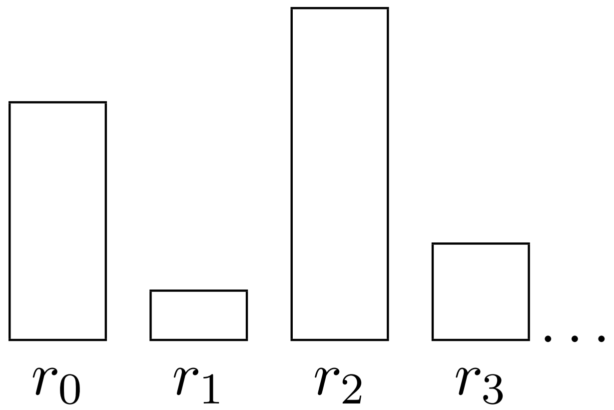
Region-based Memory Management

Question:

Can the Algol stack discipline be applied to languages with dynamic data structures and higher-order functions?

Ideas:

- Organize the heap as a stack of regions.



- At runtime, allocate all values in regions.

- Perform *region inference*: Insert allocation and deallocation directives in the program code at compile time.

A Type-and-effect System for Region-annotated Expressions

Region-annotated Terms:

The grammars for *places* (p), *values* (v), and *expressions* (e) are as follows:

$$
\begin{array}{lcl}
p & ::= & \rho \quad | \quad \bullet \\
v & ::= & d \textbf{ in } p \quad | \quad \lambda x.e \textbf{ in } p \\
e & ::= & v \quad | \quad x \quad | \quad d \textbf{ at } p \quad | \quad e_1\, e_2 \quad | \quad \lambda x.e \textbf{ at } p \\
 & | & \textbf{letregion } \rho \textbf{ in } e
\end{array}
$$

- A place of the form $\bullet$ denotes a non-existing region. Thus, access to a value stored in a place $\bullet$ signifies a reference to deallocated memory.
- Each value resides in a particular region, denoted by the "**in** p" part of the value.
- Each value-creating expression, such as d **at** ρ, is annotated with the region in which the value goes at runtime.

A Small-step Contextual Dynamic Semantics

Evaluation contexts:

$$E \quad ::= \quad [\cdot] \quad | \quad E\,e \quad | \quad v\,E \quad | \quad \textbf{letregion}\ \rho\ \textbf{in}\ E$$

Notice:

- Evaluation is allowed under **letregion**-constructs to model evaluation in the presence of non-empty region stacks.
- Reduction rules are of the form $e \leadsto e'$, which reads:

  "The expression e reduces in one step to the expression e'.

- For simplicity, we leave out a potential recursive construct:

$$e \quad ::= \quad \ldots \quad | \quad \textbf{fix}\ f(x)\ \textbf{at}\ p\ = e$$

Reduction Rules

ALLOCATION

$$\boxed{e \leadsto e'}$$

$$d\ \textbf{at}\ \rho \leadsto d\ \textbf{in}\ \rho \qquad \lambda x.e\ \textbf{at}\ \rho \leadsto \lambda x.e\ \textbf{in}\ \rho$$

DEALLOCATION:

$$\textbf{letregion}\ \rho\ \textbf{in}\ v \leadsto v[\bullet/\rho]$$

FUNCTION APPLICATION:

$$(\lambda x.e\ \textbf{in}\ \rho)\ v \leadsto e[v/x]$$

CONTEXT:

$$\frac{e \leadsto e' \quad E \neq [\cdot]}{E[e] \leadsto E[e']} \tag{3}$$

Evaluation

We define *evaluation* as the least relation formed by the reflexive transitive closure of the reduction relation $\leadsto$:

Evaluation $\boxed{e \leadsto^* e'}$

$$\frac{e \leadsto e'}{e \leadsto^* e'} \;(4) \qquad \frac{}{e \leadsto^* e} \;(5) \qquad \frac{e_1 \leadsto^* e_2 \quad e_2 \leadsto^* e_3}{e_1 \leadsto^* e_3} \;(6)$$

We further define $e \Uparrow$ to mean that there exists an infinite sequence, $e \leadsto e_1 \leadsto e_2 \leadsto \cdots$.

A Region Type System

PURPOSE: provide a type system with the guarantee

"Well-typed programs do not get stuck."

Types and type-and-places are defined by the grammars:

$$
\begin{array}{lcll}
\mu & ::= & (\tau, \rho) & \text{(Type-and-places)} \\
\tau & ::= & \mathtt{int} \mid \mu_1 \xrightarrow{\varphi} \mu_2 & \text{(Types)}
\end{array}
$$

A *type environment* (Γ) maps program variables to type and places.

Type judgments $\Gamma \vdash e : \mu, \varphi$ are read:

"In the type environment Γ, the expression e has type and place μ and effect φ."

Region Typing Rules

VALUES $\boxed{\Gamma \vdash e : \mu, \varphi}$

$$\frac{}{\Gamma \vdash d \ \textbf{in}\ \rho : (\texttt{int}, \rho), \emptyset} \quad (7)$$

$$\frac{\{x : \mu_1\} \vdash e : \mu_2, \varphi}{\Gamma \vdash \lambda x.e \ \textbf{in}\ \rho : (\mu_1 \xrightarrow{\varphi} \mu_2, \rho), \emptyset} \quad (8)$$

EXPRESSIONS:

$$\frac{}{\Gamma \vdash d \ \textbf{at}\ \rho : (\texttt{int}, \rho), \{\rho\}} \quad (9)$$

$$\frac{\Gamma + \{x : \mu_1\} \vdash e : \mu_2, \varphi}{\Gamma \vdash \lambda x.e \ \textbf{at}\ \rho : (\mu_1 \xrightarrow{\varphi} \mu_2, \rho), \{\rho\}} \quad (10)$$

$$\frac{\Gamma(x) = \mu}{\Gamma \vdash x : \mu, \emptyset} \quad (11)$$

$$\frac{\Gamma \vdash e_1 : (\mu' \xrightarrow{\varphi_0} \mu, \rho), \varphi_1 \quad \Gamma \vdash e_2 : \mu', \varphi_2}{\Gamma \vdash e_1\ e_2 : \mu, \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\rho\}} \quad (12)$$

$$\frac{\Gamma \vdash e : \mu, \varphi \quad \varphi' \supseteq \varphi}{\Gamma \vdash e : \mu, \varphi'} \quad (13)$$

$$\frac{\Gamma \vdash e : \mu, \varphi \quad \rho \notin \mathrm{frv}(\Gamma, \mu)}{\Gamma \vdash \textbf{letregion}\ \rho\ \textbf{in}\ e : \mu, \varphi \setminus \{\rho\}} \quad (14)$$

Properties of the Region Type System

Lemma (Type Preservation)

If $\vdash e : \mu, \varphi$ *and* $e \leadsto e'$ *then* $\vdash e' : \mu, \varphi$.

Lemma (Progress)

If $\vdash e : \mu, \varphi$ *then* e *is a value or* $e \leadsto e'$ *for some* e'.

The progress lemma implies that a well-typed program cannot apply a non-function to some argument or access values in regions that are deallocated.

Theorem (Type Soundness)

If $\vdash e : \mu, \varphi$ *then either* $e \Uparrow$ *or there exists some* v *such that* $e \leadsto^* v$ *and* $\vdash v : \mu, \varphi$.

Region and Effect Polymorphism

The type system extends naturally to support polymorphism in region variables and so-called *effect variables*, ranged over by ϵ.

- Effects (φ) are now sets of effect variables and region variables.
- Using effect variables, the type system can track higher-order programming with effects:
- The apply function $\lambda f.\lambda x.f\ x$ can be given the type

$$\forall \alpha\beta\rho\rho'\epsilon.((\alpha,\rho) \xrightarrow{\{\epsilon\}} (\beta,\rho')) \xrightarrow{\emptyset} (\alpha,\rho) \xrightarrow{\{\epsilon\}} (\beta,\rho')$$

Region Inference

- Region Inference can be formulated both as a *constraint-based analysis* and as a *unification-based type inference algorithm*.
- Idea: identify arrow effects using effect variables.
- Arrow types take the form $\mu \xrightarrow{\epsilon.\varphi} \mu'$.

Constraint-bases analysis (Birkedal, Tofte, Journal of The.Comp.Sc. '01):

- Whenever a type $\mu \xrightarrow{\epsilon.\varphi} \mu'$ is created, enforce the constraint $\epsilon \supseteq \varphi$.

Unification-based inference (Tofte, Birkedal, TOPLAS '98):

- When unifying two arrow effects $\epsilon.\varphi$ and $\epsilon'.\varphi'$, create a unifier:

$$S = \{\epsilon \mapsto \epsilon.\varphi'', \epsilon' \mapsto \epsilon.\varphi''\} \qquad \text{where} \qquad \varphi'' = S(\varphi) \cup S(\varphi')$$

Region Inference, Termination

Enforce consistency restrictions on introduced arrow effects.

A set of arrow effects Φ (called an *effect basis*) is said to be consistent if

1. It is functional: For all $\epsilon.\varphi \in \Phi$ and $\epsilon'.\varphi' \in \Phi$, if $\epsilon = \epsilon'$ then $\varphi = \varphi'$.

2. It is closed: For all $\epsilon.\varphi \in \Phi$ and $\epsilon' \in \varphi$, there exists φ' such that $\epsilon'.\varphi' \in \Phi$.

3. It is transitive: For all $\epsilon.\varphi \in \Phi$ and $\epsilon'.\varphi' \in \Phi$, if $\epsilon' \in \varphi$ then $\varphi' \subseteq \varphi$.

Intuitively, a consistent basis form a set of DAGs.

A *contraction* (a substitution) is defined so that it is known to “shrink a basis”.

Unification is shown to generate contractions, which, in essence results in termination of region inference.

Garbage Collecting Regions

Why combine region inference and GC?

- For non-regionized programs, adding GC reduces memory usage.
- From GC's point of view:
  - In general, region inference reduces the number of GC invocations.
  - Region-based memory management supports "almost tag-free" garbage collection.

A challenge:

- The Tofte-Talpin region typing rules permit dangling pointers!

Dangling Pointer Example

Consider the expression

$$\begin{array}{lll} e & \equiv & \textbf{letregion } \rho \\ & & \textbf{in } (\lambda y.(\lambda x.(\lambda z.(1 \textbf{ at } \rho_1) \textbf{ at } \rho_1)\ y \textbf{ at } \rho_1) \textbf{ at } \rho_1) \\ & & \quad (3 \textbf{ at } \rho) \\ & & \textbf{end} \end{array}$$

From the typing rules, we have

$$\vdash e : ((\texttt{int}, \rho_1) \xrightarrow{\{\rho_1\}} (\texttt{int}, \rho_1), \rho_1), \{\rho_1\}$$

We also have (using five reduction steps)

$$e \ \leadsto^* \ \lambda x.(\lambda z.(1 \textbf{ at } \rho_1) \textbf{ at } \rho_1)\ (3 \textbf{ in } \rho) \textbf{ in } \rho_1$$

Problem: ρ, which is in the type of y, is not in the type of $\lambda x.(\ldots)$

Disallowing Dangling Pointers

The problem with the Tofte-Talpin typing rules:

- Values stored in function closures are not required to be contained in regions mentioned in the type of the function.

$$\frac{\Gamma + \{x : \mu_1\} \vdash e : \mu_2, \varphi}{\Gamma \vdash \lambda x.e \textbf{ at } \rho : (\mu_1 \xrightarrow{\varphi} \mu_2, \rho), \{\rho\}} \quad (15)$$

The solution:

- Enforce region variables in the type of free variables of a function to appear in the type of the function.

$$\frac{\Gamma + \{x : \mu_1\} \vdash e : \mu_2, \varphi \quad \forall y \in \mathrm{fv}(\lambda x.e).\mathrm{frv}(\Gamma(y)) \subseteq \mathrm{frv}(\mu_1 \xrightarrow{\varphi} \mu_2, \rho)}{\Gamma \vdash \lambda x.e \textbf{ at } \rho : (\mu_1 \xrightarrow{\varphi} \mu_2, \rho), \{\rho\}} \quad (16)$$

- The restriction has little impact on memory usage in practice (Elsman, TLDI'03).

Cheney's Algorithm for Regions

Extend Cheney's copying garbage collection algorithm to work with regions (Hallenberg, Elsman, Tofte, PLDI'02).

- Perform a Cheney copying collection on each region on the region stack.
- If a live value resides in a region r before a collection, the value must reside in the same region r after the collection.

HOW IT WORKS:

- All values reachable from the root set are evacuated into "to-space" region pages associated with the region.
- After a collection, all "from space" region pages are inserted into the free list of pages.

Example: Bootstrapping the MLKit

Compiling the MLKit with MLton produces an executable **Kit1** (takes 12min).

When running, **Kit1** uses the MLton runtime system.

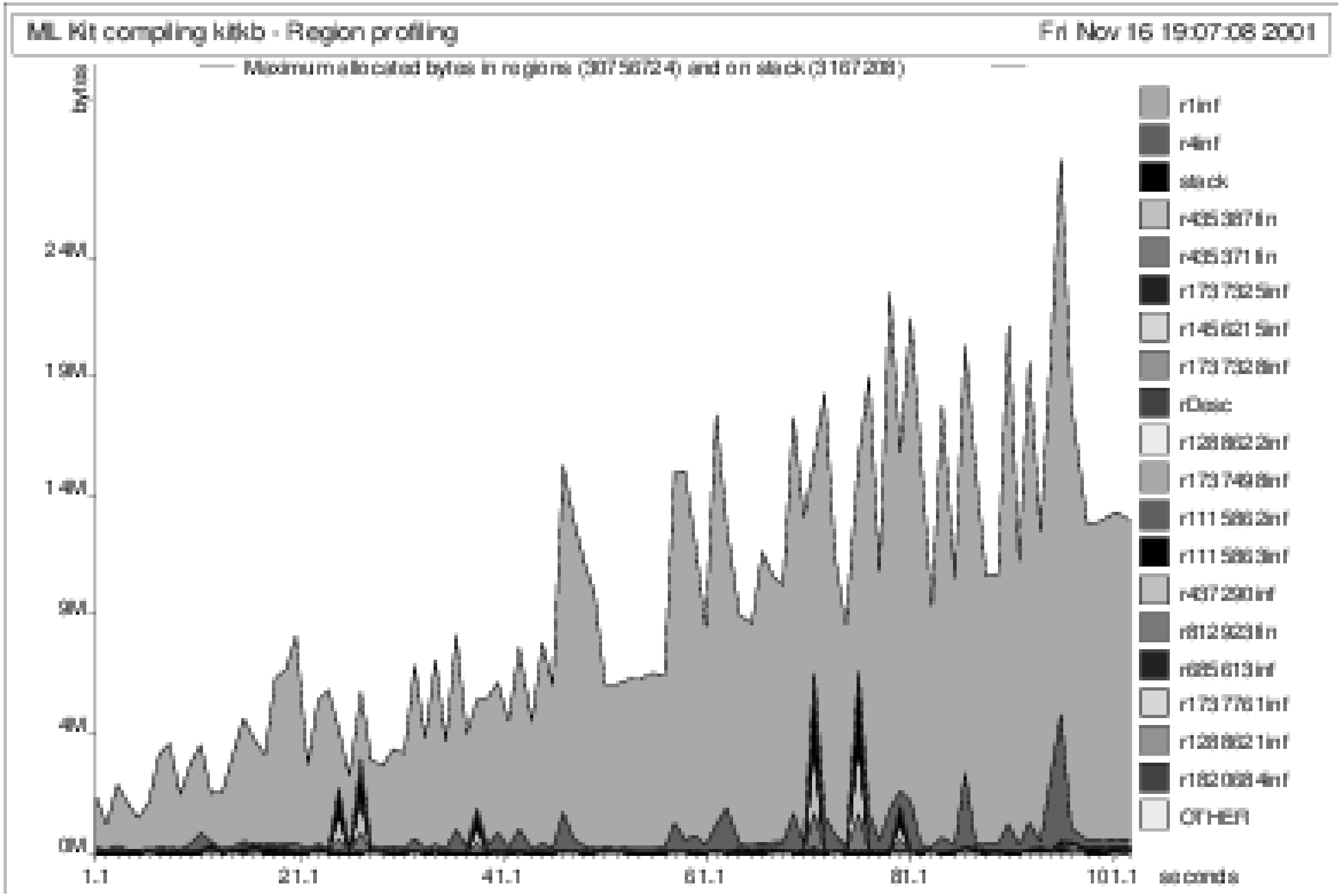
EXPERIMENTS:

- Using **Kit1** to compile the MLKit takes 3min and results in an executable **Kit2**, which uses a runtime system that combines regions and garbage collection.*
- Using **Kit2** to compile the MLKit takes 6min.
- But **Kit1** and **Kit2** are not whole-program compilers!

*The experiments were run on a MacBook Pro 2,7GHz Intel Core i7 laptop with 16Gb RAM.

Garbage Collecting Regions in Practice

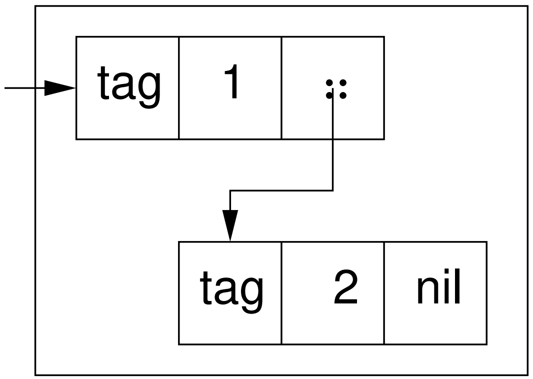
EXAMPLE: A region profile of running the bootstrapped MLKit with the Knuth-Bendix test program as input:



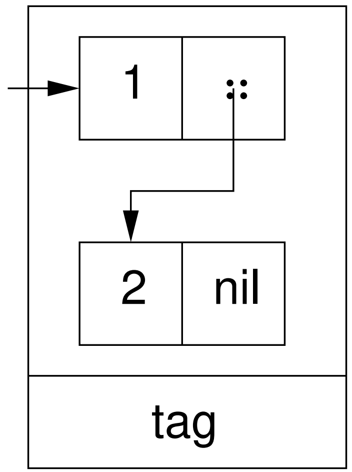
Almost Tag-free Garbage Collection

Refine the region typing rules to disallow values of different types to reside in the same region.

- Runtime tags can then be moved from individual values to the region in which each value is stored.
- Dramatic savings in heap usage can be obtained, particularly for lists and tree-like data structures.



Untagged region
with tagged values



Tagged region
with untagged values

Possible Future Work

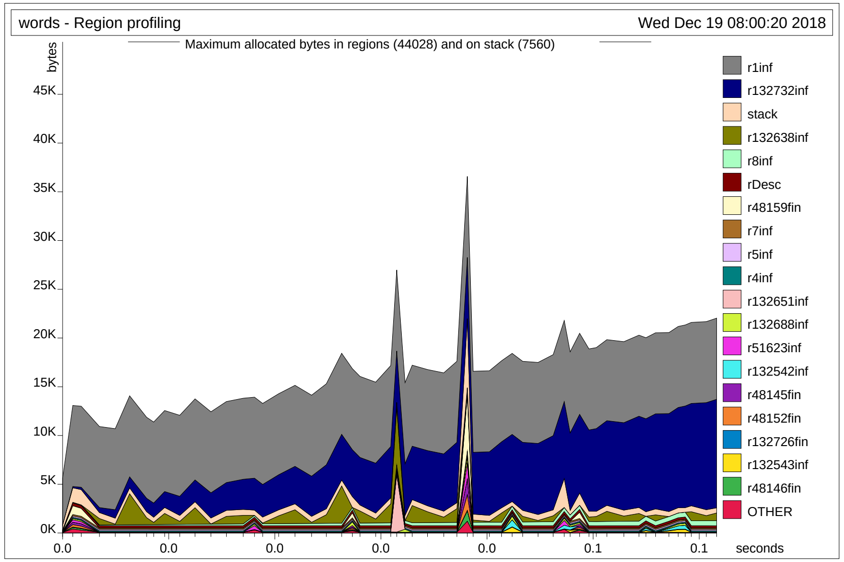
- Programming explicitly with regions
- Combining region inference and generational collection
- Thread support
- Region type systems for low-level languages (e.g., bytecode)

Exercises

① Clone the git-repository

`https://github.com/melsman/effects-seminar-public.git`

② Write a program that, in Dicken's Christmas Carol (available from the `texts/` folder), will find the number of sentences containing pairs of permuted words. Get the program to run in as little space as possible. Here is what you should aim for:



③ Prove the type soundness theorem on slide 3-9.