The MLKit with Regions— Lecture 2/3

Martin Elsman, DIKU

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

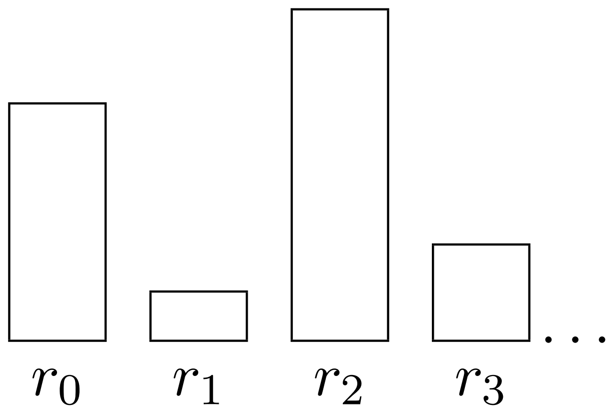
- Region-based Memory Management
- History of region-based memory management in the MLKit
- Supportive claims:
  - Programs can be compiled to run efficiently.
  - Region inference integrates well with complex language constructs, such as Standard ML modules.
  - It scales to large programs.
- Related and Future work
- 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.

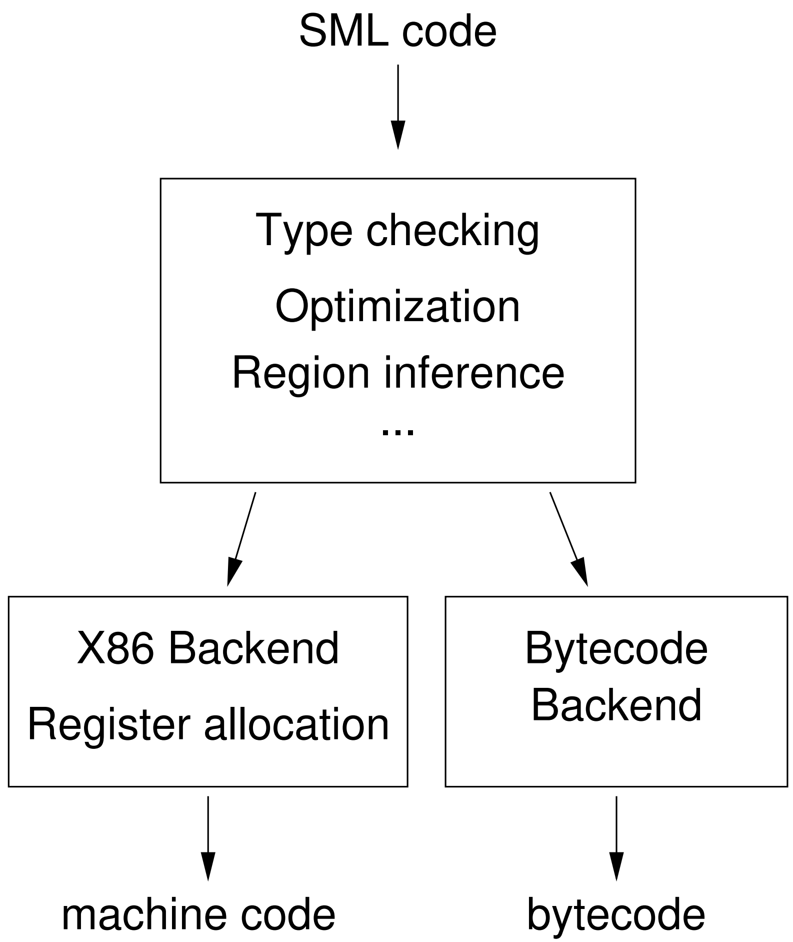
- Perform *region inference*: Insert allocation and deallocation directives in the program code at compile time.

The MLKit Compiler

A compiler for the **full** Standard ML programming language.

Properties of the MLKit:

- It is based on region inference.
- All annotations are inferred automatically—no directives need be annotated by the programmer.
- It generates efficient X86 machine code or portable bytecode.



Many people have been involved:

Mads Tofte, Lars Birkedal, Niels Hallenberg, Tommy Olesen, Peter Sestoft, ...

Outline of the Talk

- A history of region-based memory management in the MLKit
- Supportive claims:
  - Programs can be compiled to run efficiently.
  - Region inference integrates well with complex language constructs, such as Standard ML modules.
  - It scales to large programs.
- Applications
  - Combining region inference and garbage collection
  - SMLserver—an efficient region-based multi-threaded Web server platform for Standard ML programs
- Related and Future work

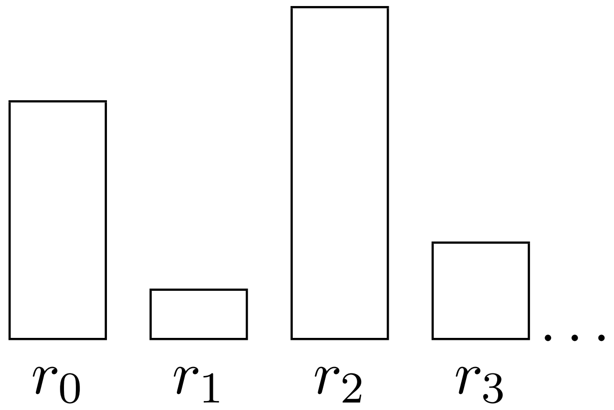
Region-based Memory Management

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.

History: The Basics

THE BASICS (TOFTE AND TALPIN, POPL'94):

- Source language:

$$e \; ::= \; x \mid d \mid \lambda x.e \mid e_1\, e_2 \mid \mathtt{letrec}\; f\,(x) = e_1 \;\mathtt{in}\; e_2 \;\mathtt{end}$$

- Every well-typed source language expression e can be translated into a region-annotated expression e'.

- Possible annotations:

  $\boxed{\mathtt{letregion}\;\rho\;\mathtt{in}\;e\;\mathtt{end}}$ At runtime, first a region r is allocated and bound to the region variable ρ. Then e is evaluated, probably using r. Finally, r is deallocated.

  $\boxed{e\;\mathtt{at}\;\rho}$ Here e is an allocating expression and ρ is a region variable, which denotes a region at runtime.

A Simple Region-annotated Expression

Consider a region-annotated language extended with pairs...

EXAMPLE:

$$
\begin{array}{l}
\texttt{let } y = \\
\quad \texttt{letregion}\, \rho, \rho' \texttt{ in} \\
\qquad \texttt{let } x = \texttt{(}3 \texttt{ at } \rho \texttt{,} 4 \texttt{ at } \rho'' \texttt{)} \texttt{ at } \rho' \\
\qquad \texttt{in snd}\, x \\
\qquad \texttt{end} \\
\quad \texttt{end} \\
\texttt{in ...} \\
\texttt{end}
\end{array}
$$

- The pair bound to y is located in region ρ'', which is global in this program.
- After the value (to be bound to y) is computed, regions ρ and ρ' can be safely deallocated.

History: Region Polymorphism

ANNOTATIONS:

- $\texttt{letrec}\; f[\rho_1, \cdots, \rho_n]\,(x) = e \;\texttt{in}\; e' \;\texttt{end}$
- $f[\rho_1, \cdots, \rho_n]$

TYPES:

- $f : \forall \rho.\texttt{unit} \xrightarrow{\{\rho\}} (\texttt{int}, \rho)$

$$
\begin{array}{l}
\texttt{letrec}\; f[\rho]\,() = 5 \;\texttt{at}\; \rho \\
\texttt{in} \ldots \texttt{letregion}\; \rho' \;\texttt{in} \\
\qquad\quad \ldots f[\rho']\,() \;\ldots \\
\qquad \texttt{end} \\
\texttt{end}
\end{array}
$$

WITHOUT REGION-POLYMORPHISM:

- All calls of f return their result in the same region.
- The region ρ is kept live until no result of f is needed.

$$
\begin{array}{l}
\texttt{letregion}\; \rho \;\texttt{in} \\
\;\texttt{letrec}\; f\,() = 5 \;\texttt{at}\; \rho \\
\;\texttt{in} \\
\qquad f\,() \;\ldots\; f\,() \\
\;\texttt{end} \\
\texttt{end}
\end{array}
$$

History: Region Polymorphic Recursion

FEATURE:

Each recursive function invocation may use its own local region!

- $\texttt{fac} : \forall\rho.(\texttt{int},\rho) \xrightarrow{\{\rho\}} (\texttt{int},\rho)$
- This type is given to `fac` both for external uses of *fac* **and inside the body** of the function definition.

WITHOUT POLYMORPHIC RECURSION:

The result of a recursive call is stored in the same region as the result of the function.

```
letrec fac[ρ](n) =
 if n=0 then 1 at ρ
 else
  letregion ρ' in
   (n*fac[ρ'](n-1)) at ρ
  end
in fac[ρ0] 100 end
```

```
letrec fac[ρ](n) =
 if n=0 then 1 at ρ
 else
  (n*fac[ρ](n-1)) at ρ
in fac[ρ0] 100 end
```

History: Storage Mode Analysis

Solves problem with tail recursion:

Allow values to be stored “at the bottom” of a region if it can be established that no value in the region is live.

Example:

```
letrec sumit [ρ] (p:int*int) =
  let acc = #1 p
      n = #2 p
  in if n=0 then p
     else sumit [ρ] ((n+acc, n-1) atbot ρ)
  end
in #1(sumit [ρ'] ((0,100) atbot ρ')) end
```

Enables a programmer to write region-friendly programs.

History: On Safety and Correctness

The Tofte-Talpin translation is formalized as a set of region inference rules, allowing inference of sentences of the form *TE* $\vdash e \Rightarrow e' : (\tau, \rho), \varphi$.

A correctness proof, which relates the evaluation of the source and target expressions, was established using co-induction.

MUCH RELATED WORK IN THIS AREA HAS FOLLOWED:

- Crary, Walker, Morrisett (POPL'99)
- Helsen and Thiemann (HOOTS'00)
- Calcagno (POPL'01)
- Calcagno, Helsen, and Thiemann (*Inf. and Comp.*, 2002)

History: Initial Experiments with the MLKit

Initial experiments — 1995:

- Small programs were compiled into machine code via C.
- The programs ran slower and used more memory than when compiled with other systems!

Questions:

① What is the administrative overhead of regions?

② What optimizations guarantee that an optimized program uses no more memory than the unoptimized program?

③ Does the technology scale to Standard ML modules?

④ Does the technology scale to large programs?

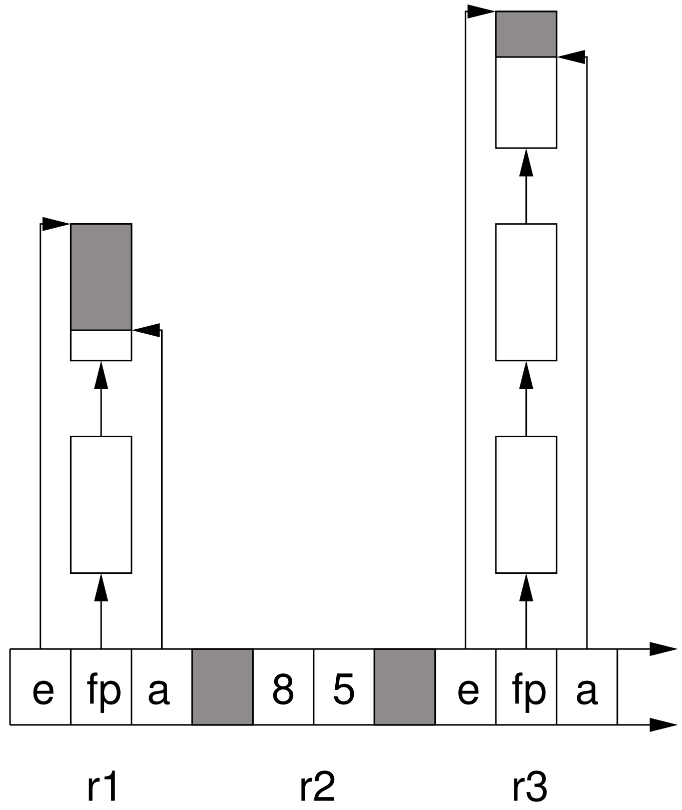
⑤ Is region inference alone always a good idea?

On The Administrative Overhead of Regions

Multiplicity Analysis (Birkedal et. al, POPL'96):

A type-based *multiplicity analysis* distinguishes between finite regions and infinite regions:

- **Finite regions** are represented directly on the runtime stack.
- **Infinte regions** are lists of fixed-sized region pages.



On average, 90 percent of all regions hold only one value!

Values that do not fit in a region page, such as large arrays, are malloc'ed.

On The Administrative Overhead of Regions

MINIMIZE REGION POLYMORPHISM WHEN POSSIBLE:

Specialize a function with respect to a region parameter ρ, if

① The function is local.

② Recursive calls pass ρ as argument.

③ Other applications are called with the same actual argument (ρ').

④ The region ρ' is in scope at the declaration of the function.

$$
\begin{array}{l}
\texttt{letrec f } [\rho]\ \texttt{x} = \\
\quad \cdots\ \texttt{f } [\rho]\ e \\
\quad \cdots\ (\texttt{7 at } \rho) \\
\quad \cdots \\
\texttt{in } \cdots\ \texttt{f } [\rho']\ (\texttt{4 at } \rho'') \\
\qquad \cdots\ \texttt{f } [\rho']\ (\texttt{9 at } \rho'') \\
\qquad \cdots \\
\texttt{end}
\end{array}
$$

For some programs, this optimization has a 10 percent effect on execution time.

On The Administrative Overhead of Regions

Unboxing of datatypes:

- Word-sized values, like integers and enumerations, are represented unboxed and are therefore not stored in regions (unless they are part of another region-allocated value).

- Datatypes with 3 or less unary constructors with boxed arguments are represented unboxed, using the least significant bits in pointers.

Unboxing of function arguments:

- When possible, Curried functions and functions taking a tuple as argument are translated into multi-argument functions.

These optimizations dramatically decrease the number of region variables in a program.

On Efficiency

Questions:

- Can programs be compiled to run efficiently?
- Which traditional optimizations can be applied?

Necessary property of optimizations:

- The optimized program should use no more memory than the unoptimized program!

Many optimizations are possible:

- Function inlining
- Function specialization
- Elimination of records
- Uncurrying
- Unboxing function arguments
- Fix-minimization

Region-unsafe Optimizations

Common subexpression elimination:

- Two otherwise distinct regions may be forced to be identical.

Constant folding:

- The region typing rule for conditional expressions forces the types of branches to be identical.

```
letrec f(x) =
  if true then x
  else f(x)
in ... end
```

- The programmer may use bogus conditional expressions to ensure proper tail recursion.

- Constant folding may eliminate such conditionals and cause the program to use asymptotically more memory.

Integration with Common Language Constructs

ML DATATYPES:

- Two distinct values of the same datatype can be stored in different regions.

REFERENCES:

- Region inference works well with references, although frequent updates may result in non-released memory.

EXCEPTION CONSTRUCTS:

- Raising and handling exceptions integrate well with the region stack.
- Unary exception constructors are problematic.

Modules and Separate Compilation

Static interpretation of modules — Elsman (ICFP'99):

- Regard the module language as a linking language for building a complete program from a set of program fragments.
- Type check functors when they are declared, but postpone code generation until the functor is applied.

Separate compilation:

- Necessary for compilation to be feasible.
- Expose information about exported identifiers (e.g., region types) in export interfaces.
- When a program fragment p is compiled, make use of export interfaces of program fragments on which p depends.

Avoiding Unnecessary Recompilation

Static interpretation of modules is feasible only if most recompilation is avoided upon modification of source code.

Cut-off recompilation:

- A program fragment (e.g., a functor body) is not necessarily recompiled if program fragments on which it depends have changed.
- Recompilation is triggered only if information about dependent identifiers (e.g., the region type of an identifier) has changed.

Correctness property:

- Upon modification of source code, the recompilation scheme results in an executable identical to the result of compiling the whole program from scratch.

So: Does Region-Inference Work in Practice?

THE MLKIT:

A compiler for the **full** Standard ML programming language.

- The MLKit is based on region inference.
- All annotations are inferred automatically—no directives need be annotated by the programmer.
- It generates efficient X86 machine code or portable bytecode.

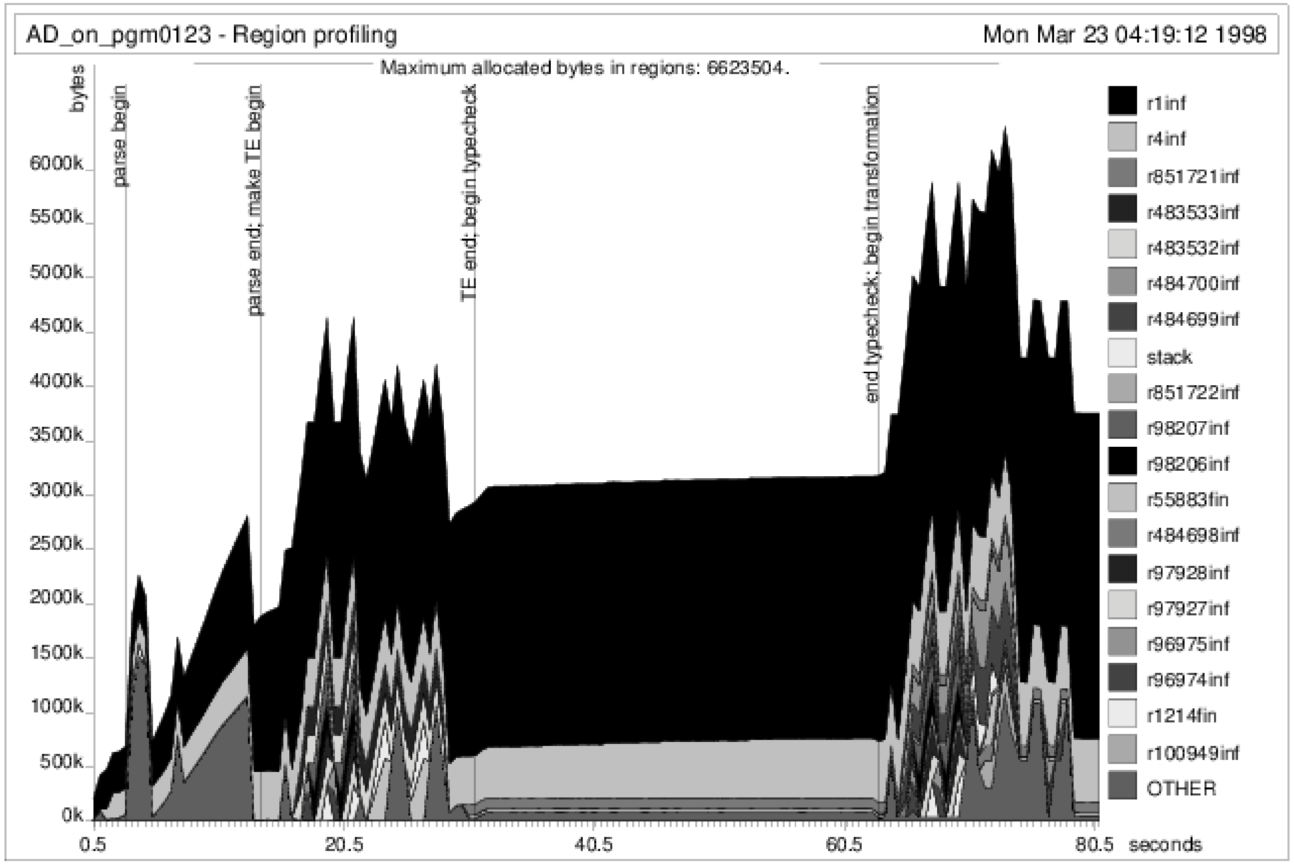
POSSIBLE PROGRAMMER FEEDBACK:

- Inferred region annotations may be inspected by the programmer.
- A region profiler may be used by the programmer to tell which regions are big at runtime.

Yes — Region Inference Works for Large Programs!

Example: AnnoDomini Y2K analysis tool:

A region profile of running AnnoDomini, on an example Cobol program:



Related Work, not mentioned so far

Region-based Memory Management:

- Safe programming of low-level code
  - Cyclone (Grossman et. al, PLDI'02)
  - Vault (Fähndrich, DeLine, PLDI'01, PLDI'02)
  - Types for Crash Prevention (L. Pareto, PhD thesis, Chalmers '2000)
- Dynamic region systems — Gay and Aiken (PLDI'01)
- Improvement of region annotations — Aiken et. al (POPL'95)
- Other region type systems — Henglein, Makholm, Niss (PPDP'01)

SMLserver

An efficient region-based multi-threaded Web server platform for Standard ML programs (Elsman, Hallenberg, PADL'03).

Properties:

- Region-based memory management works well for programs that run quickly but are executed often. No GC!
- Cache locality: Two simultaneously running scriptlets may use the same region page at different times during execution.

Other Features:

- Static typing guarantees that scriptlets respond with conforming XHTML.
- Forms are statically typed: A form targeting a scriptlet is consistent with the scriptlet's use of form data.

Compiling with MLKit

① Install MLKit from either Homebrew for Mac OS (`brew install mlkit`) or the debian package available from:

`https://launchpad.net/~pmunksgaard/+archive/ubuntu/mlkit`

② Copy the example directory `kitdemo` to your own home directory:

```
$ find /usr/local -name 'kitdemo'
/usr/local/Cellar/mlkit/4.3.15/share/mlkit/kitdemo
$ cp -a /usr/local/Cellar/mlkit/4.3.15/share/mlkit/kitdemo ~/kitdemo
```

③ Compile and run the `helloworld` program using MLKit:

```
$ cd ~/kitdemo
$ mlkit -no_gc -o helloworld helloworld.sml
...
[wrote executable file: helloworld]
$ ./helloworld
hello world
```

Alternatively, download and install the latest version of MLKit from the MLKit github repository `http://github.com/melsman/mlkit`.

Example: Compiling the `pairgen` function

① Assume that the function

```
fun pairgen (a:int) = (a,a)
```

is stored in a file `pairgen.sml`.

② To compile the file with flags for printing types and effects in printed intermediate programs, the following command is used:

```
$ mlkit -Ptypes -Peffects -Pcee pairgen.sml
```

③ The MLKit reports the following type scheme for the `pairgen` function:

```
all r69,e70.int-e70(U(put(r69)))->(int*int,r69)
```

④ The function is parametric in a region variable `r69` and an *effect variable* `e70`.

⑤ When the function is applied, it “puts” values in the region passed in for `r69`.

⑥ Effect variables are technically necessary to refer to effects in the case of region (and effect) polymorphism.

Profiling with the MLKit

To generate a region profile for the `life.sml` program, proceed as follows:

- Compile the file `life.sml` with profiling enabled:

```
$ mlkit -prof life.sml
```

- Run the program:

```
./run -microsec 100
```

- Produce a postscript-file from the generated profiling data:

```
$ rp2ps -region -name Life
```

- Show the region profile:

```
$ ps2pdf region.ps region.pdf; open region.pdf
```

Exercises

① Try the examples on the previous slides.

② Determine the region type schemes for the following functions:

```
fun plus a b : real = a + b
fun swap1 (a,b) = (b,a)
fun swap2 (a,b) = if false then (a,b) else (b,a)
fun pairgen a b = (a,b)
fun apply f x = f x
fun op o (f,g) x = f(g x)
```

Use the MLKit (with command line options `-Ptypes -Peffects -Pcee`) to test your solutions:

```
$ mlkit -Ptypes -Peffect -Pcee myfuns.sml
```

③ Investigate the region-annotated version of the `fromto` function in Chapter 6 of the MLKit manual. Without region polymorphism, what type could we assign to the `fromto` function?