Parametric Prediction of Heap Memory Requirements

Víctor Braberman, Federico Fernandez, **Diego Garbervetsky**, Sergio Yovine*

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires (UBA)
(*) VERIMAG, France. Currently visiting UBA.

Motivation

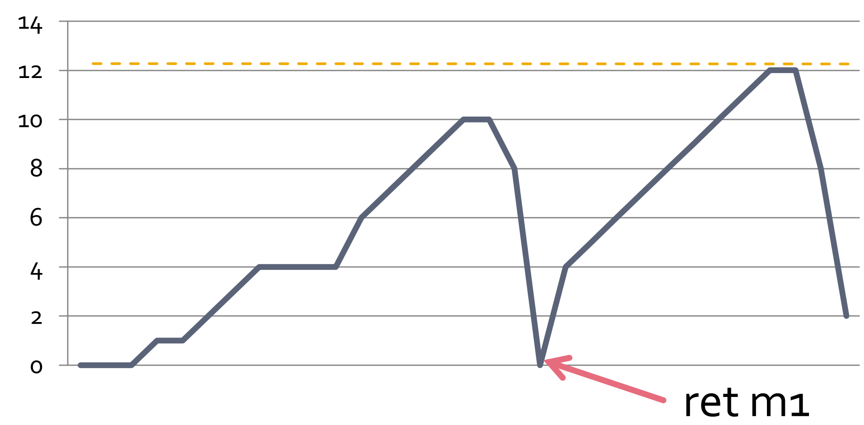
- Context: Java like languages
  - Object orientation
  - Automatic memory management (GC)
- Predicting amount of memory allocations is very hard
  - Problem undecidable in general
    - Impossible to find an exact expression of dynamic memory requested, even knowing method parameters
- Predicting actual memory requirements <u>is harder</u>
  - Memory is recycled
    - Unused objects are collected
    - $\Rightarrow$ memory required <= memory requested/allocated

Example

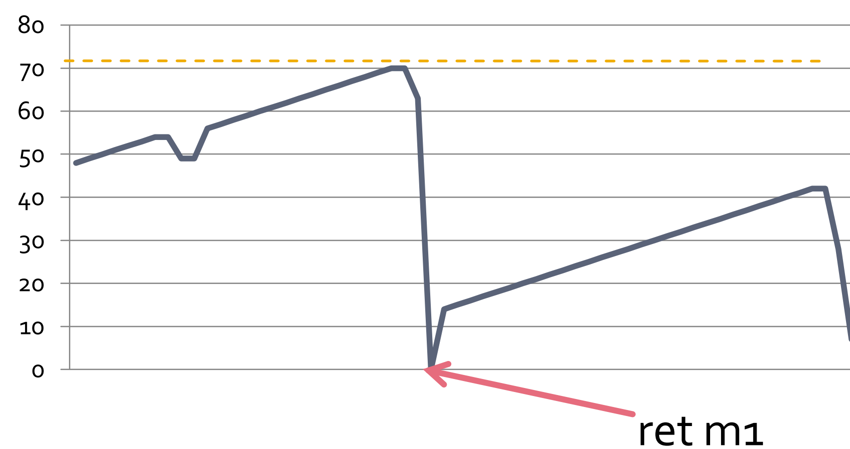
How much memory is required to run mo?

```
void m0(int mc) {
1: m1(mc);
2: B[] m2Arr=m2(2 * mc);
}
void m1(int k) {
3: for (int i = 1; i <= k; i++){
4:    A a = new A();
5:    B[] dummyArr= m2(i);
   }
}
B[] m2(int n) {
6: B[] arrB = new B[n];
7: for (int j = 1; j <= n; j++) {
8:    arrB[j-1] = new B();
9:    C c = new C();
10:   c.value = arrB[j-1];
   }
11: return arrB;
}
```

“Ideal” consumption mo(2)



Ideal consumption mo(7)



Our goal

- An expression over-approximating the peak amount of memory consumed using an ideal memory manager
  - **Parametric**
  - Easy to evaluate
    - E. g. :Required(m)(p1,p2) = $2p^2 + p1$
  - Evaluation cost known "a priori"

Given a method `m(`p_1`,..,`p_n`)`

- `peak(m)` : an expression in terms of $p_1, \ldots, p_n$ for the max amount of memory consumed by `m`

Context

- Previous work
  - A general technique to find non-linear parametric upper- bounds of dynamic memory allocations
    - **totAlloc(m)** computes an expression in terms of *m* parameters for the amount of dynamic memory requested by any run starting at *m*
    - Relies on programs invariants to approximate number of visits of allocating statements
  - Using a scope-based region management…
    - An application of that technique to approximate region sizes

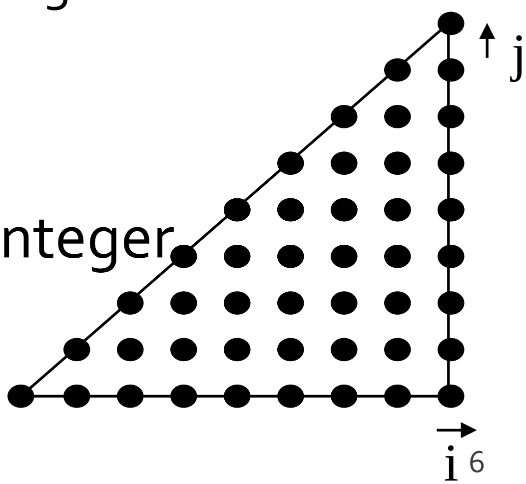
Computing dymamic memory allocations

Basic idea: counting visits to memory allocating statements.

```
for(i=0;i<n;i++)
   for(j=0;j<i;j++)
      • new C()
```

$\phi \equiv \{0 \leq i < n, 0 \leq j < i\}$: a set of constraints describing a iteration space

- Dynamic Memory allocations ≅ number of <u>visits</u> to new statements
- ≅ number of <u>possible variable assignments</u> at its control location
- ≅ <u>number of integer solutions</u> of a predicate constraining variable assignments at its control location (i.e. an <u>invariant</u>)



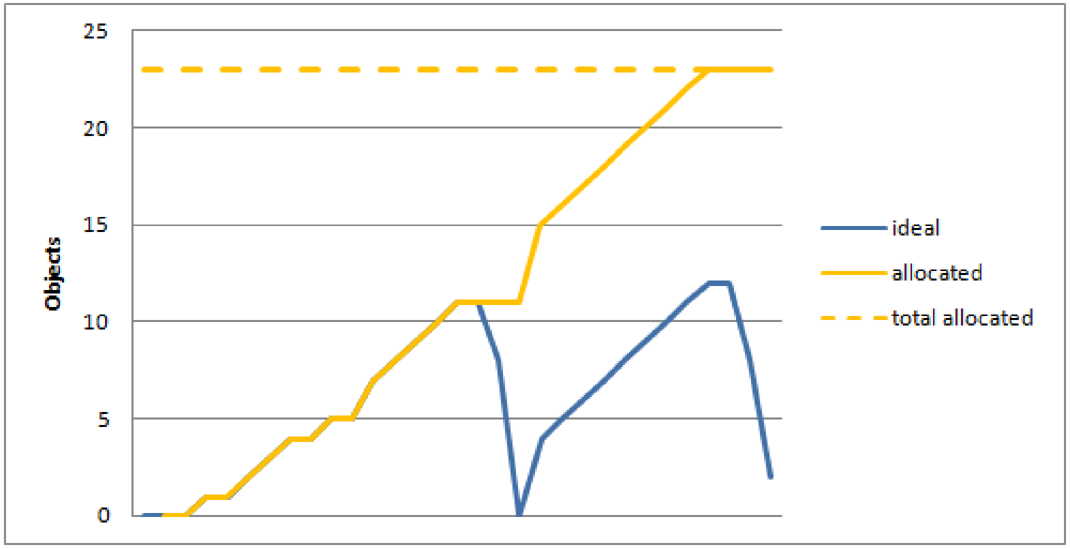
For linear invariants, # of integer solutions = # of integer points = Ehrhart polynomial size(B) * (½**k**2+½**k**)

Memory requested by a method

- How much memory (in terms of mo parameters) is <u>requested/allocated</u> by mo

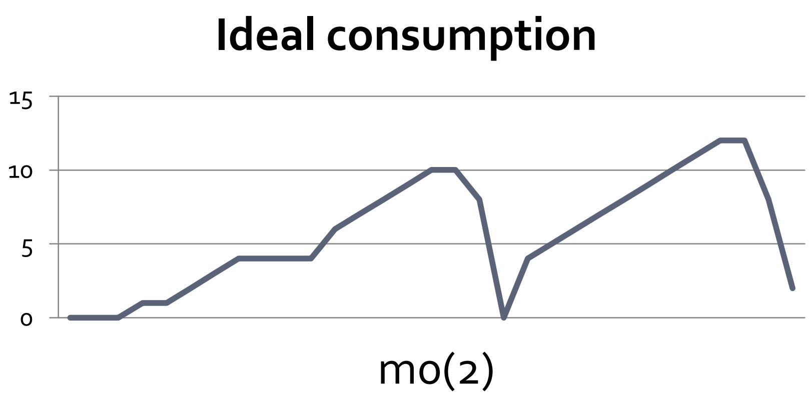
$$\text{totAlloc(mo)(mc)} = \sum_{cs \in CS\_m0} S(m0, cs)$$

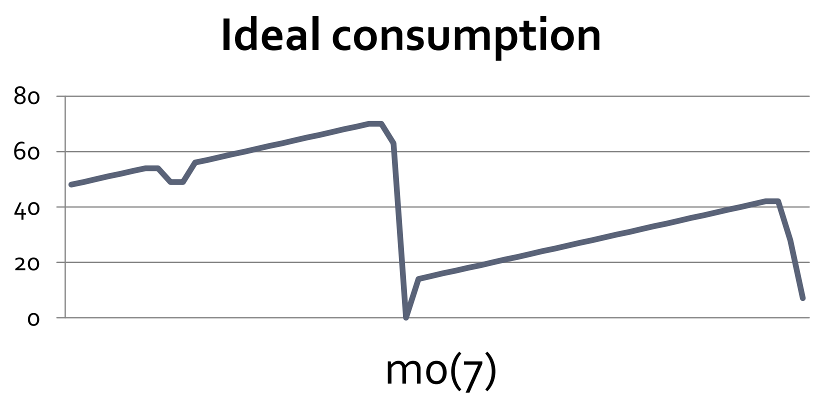
$= (\texttt{size(B[])} + \texttt{size(B)} + \texttt{size(C)})\mathbf{(1/2\ mc^2 + 5/2\ mc)}$
$+ \texttt{size(A)}\ \mathbf{mc}$



Problem

- Memory is released by a garbage collector
  - Very difficult to predict **when**, **where**, and **how** many object are collected



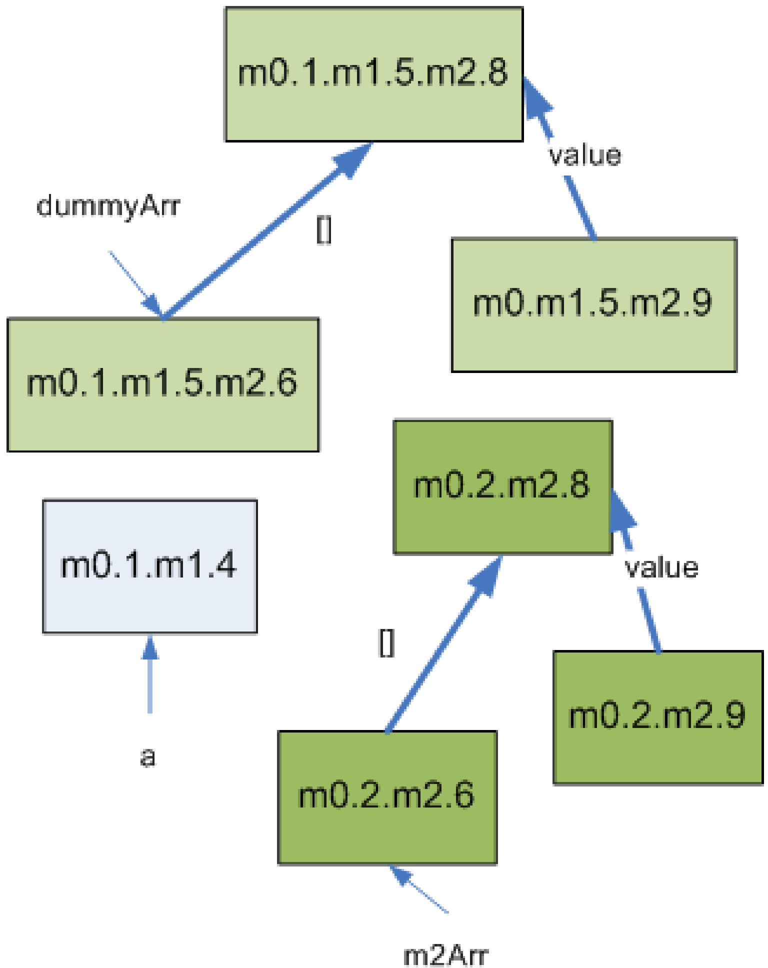


- Our approach: Approximate GC using a scope-based region memory manager

Region-based memory management

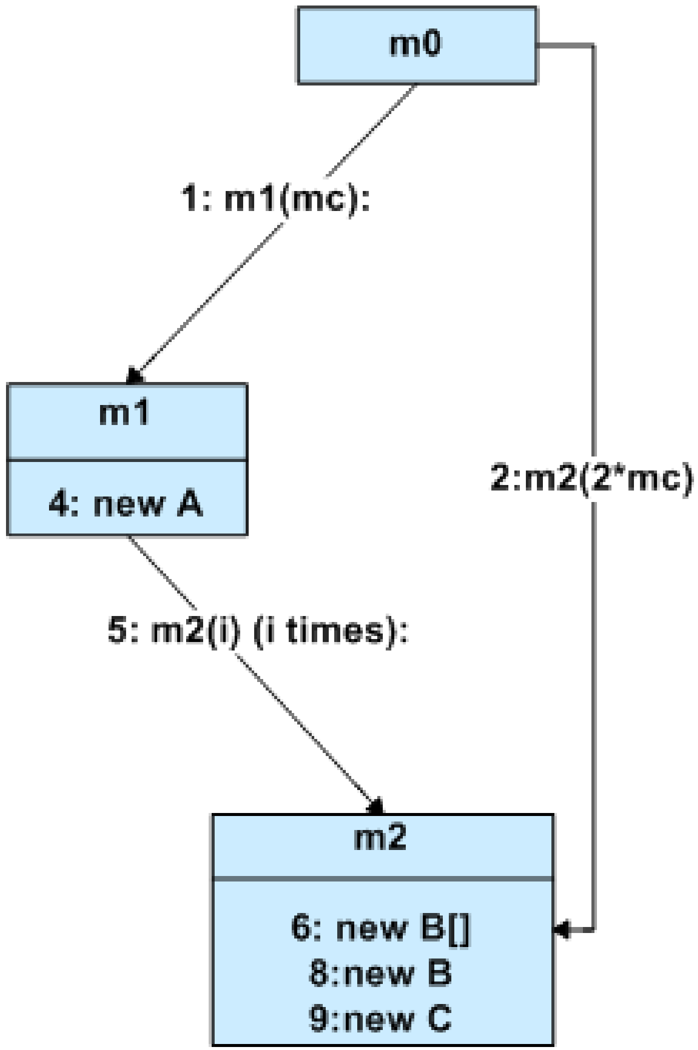
Memory organized using m-regions

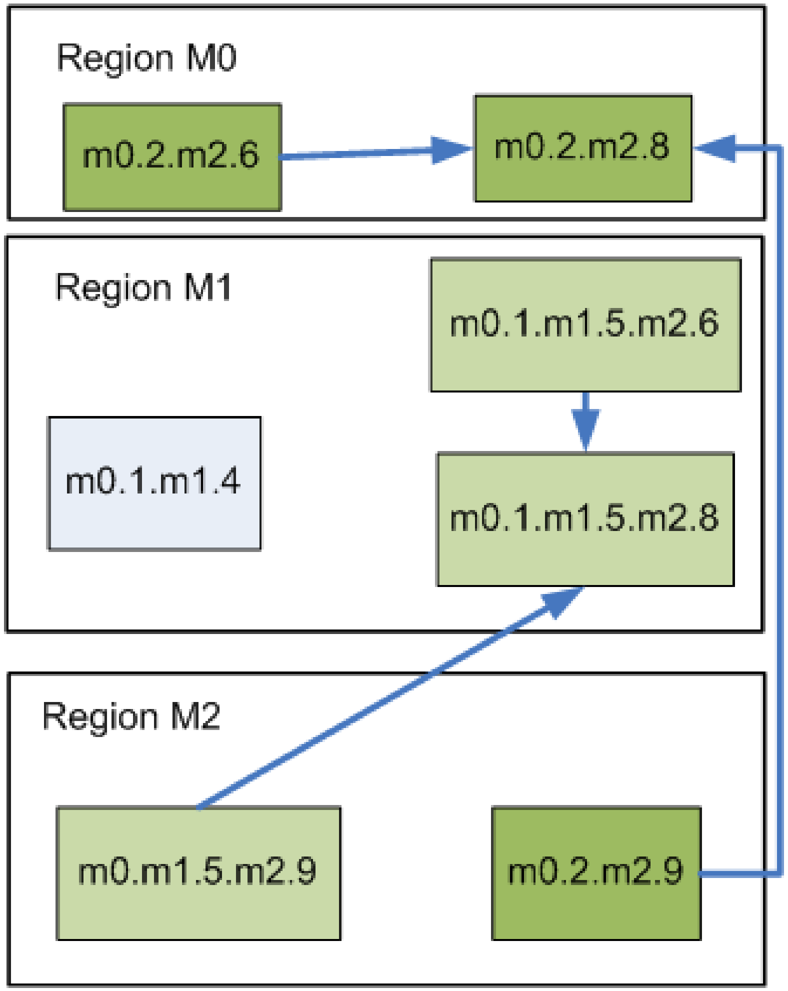
```
void m0(int mc) {
1: m1(mc);
2: B[] m2Arr=m2(2 * mc);
}
void m1(int k) {
3: for (int i = 1; i <= k; i++){
4:    A a = new A();
5:    B[] dummyArr= m2(i);
   }
}
B[] m2(int n) {
6: B[] arrB = new B[n];
7: for (int j = 1; j <= n; j++) {
8:    arrB[j-1] = new B();
9:    C c = new C();
10:   c.value = arrB[j-1];
   }
11: return arrB;
}
```



Region-based memory management

Memory organized using **m-regions**





Region-based memory management

Escape Analysis

```
void m0(int mc) {
1: m1(mc);
2: B[] m2Arr=m2(2 * mc);
}
void m1(int k) {
3: for (int i = 1; i <= k; i++){
4:    A a = new A();
5:    B[] dummyArr= m2(i);
   }
}
B[] m2(int n) {
6: B[] arrB = new B[n];
7: for (int j = 1; j <= n; j++) {
8:    arrB[j-1] = new B();
9:    C c = new C();
10:   c.value = arrB[j-1];
   }
11: return arrB;
}
```

- Escape(m): objects that live **beyond** m
  - Escape(m0) = {}
  - Escape(m1) = {}
  - Escape(m2) = {m2.6, m2.8}
- Capture(m): objects that do **not** live more that m
  - Capture(m0) = {m0.2.m2.6, m0.2.m2.8},
  - Capture(m1) = {m1.4, m0.1.m1.5.m2.6, m0.1.m1.5.m2.8},
  - Capture(m2) = {m2.9}
- Region(m) ≅ Capture(m)

Obtaining region sizes

- Region(m) ≅ Capture(m)
- `memCap(m)` : an expression in terms of $p_1, \ldots, p_n$ for the amount of memory required for the region associated with `m`
- `memCap(m)` is `totAlloc(m)` applied only to captured allocations

- `memCap(m0)` = **(size(B[]) + size(B)).2mc**
- `memCap(m1)` = **(size(B[]) + size(B)).(1/2 k^2 +1/2k) +size(A).k**
- `memCap(m2)` = **size(C).n**

Approximating peak consumption

Approach:

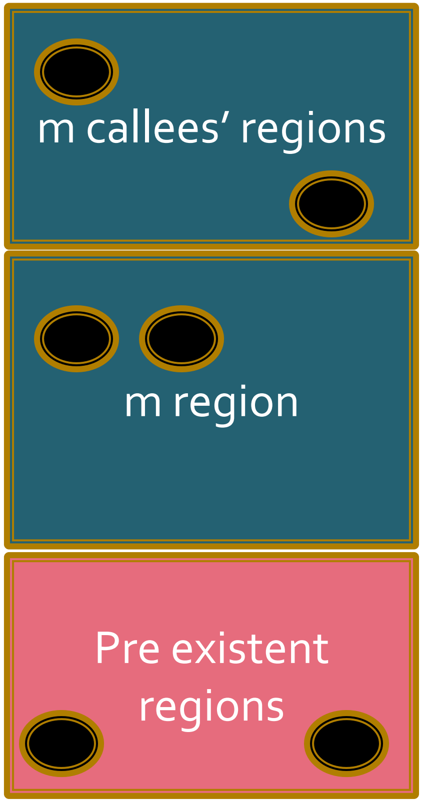
- Over approximate an ideal memory manager using an scoped-based memory regions
  - m-regions: one region per method
- **When & Where:**
  - created at the beginning of method
  - destroyed at the end
- **How much** memory is allocated/deallocated in each region:
  - `memCap(m)` >= actual region size of m for any call context
- **How much** memory is allocated in outer regions :
  - `memEsc(m)` >= actual memory that is allocated in callers regions

Approximating peak consumption

- **Peak(m) = Peak↑(m) + Peak↓(m)**
  - peak↑(m): peak consumption for objects allocated in regions created when m is executed
  - peak↓(m): consumption for objects allocated in regions that already exist before m is executed

Our technique:

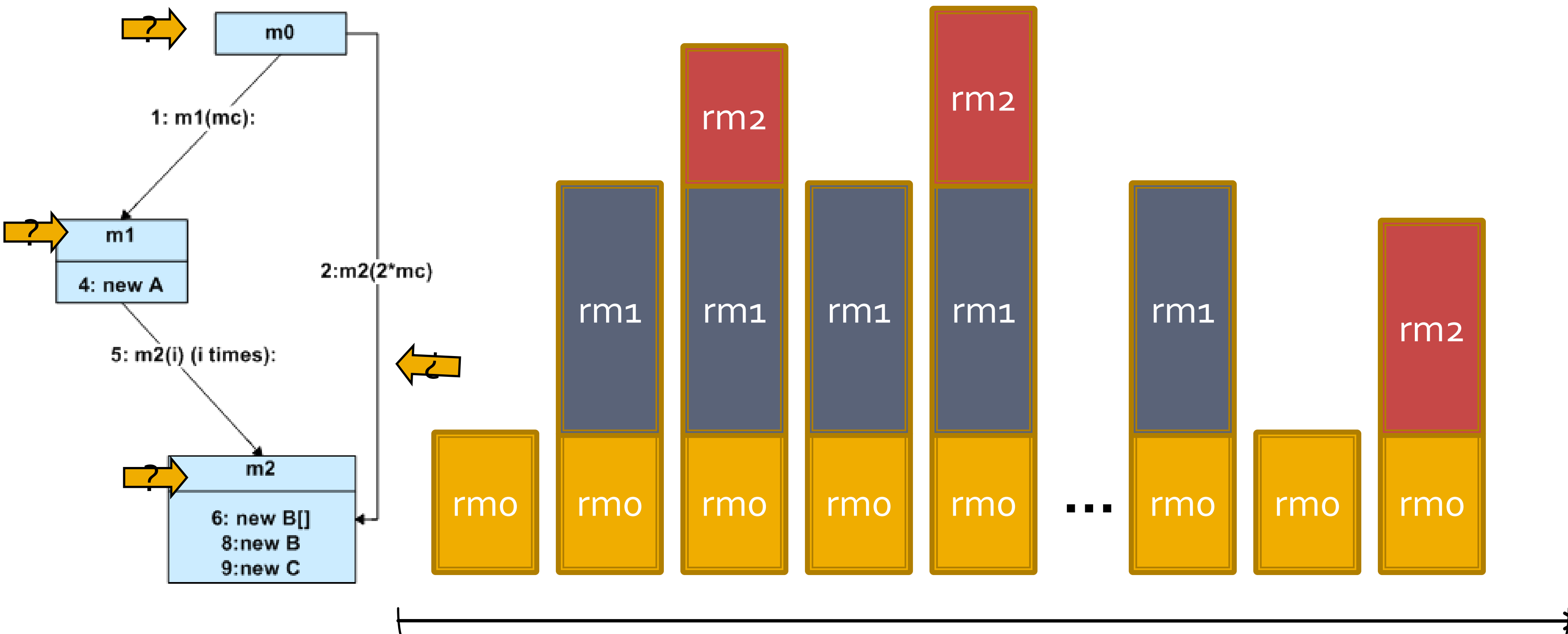
- **mem ↑(m) >= Peak ↑(m)**
  - Approximation of peak memory allocated in newly created regions
- **mem↓(m) >= Peak↓(m)**
  - Approximation of memory allocated in preexistent regions (memEsc(m))



Approximating Peak↑(m)

Region's stack evolution

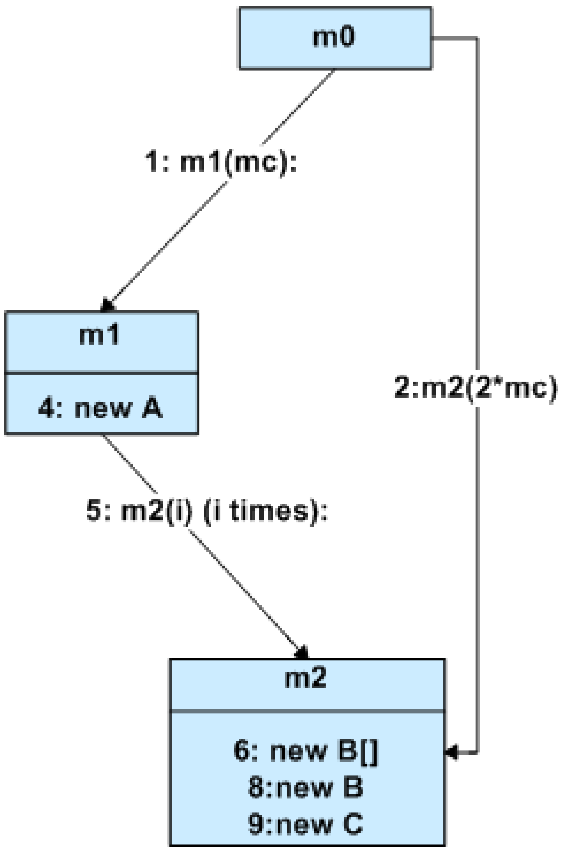
- Some region configurations can not happen at the same time



peak↑(σ_o, mo) = max Σ size($r_k(\sigma)$)

Approximating Peak↑(m)

Region sizes may vary according to method calling context



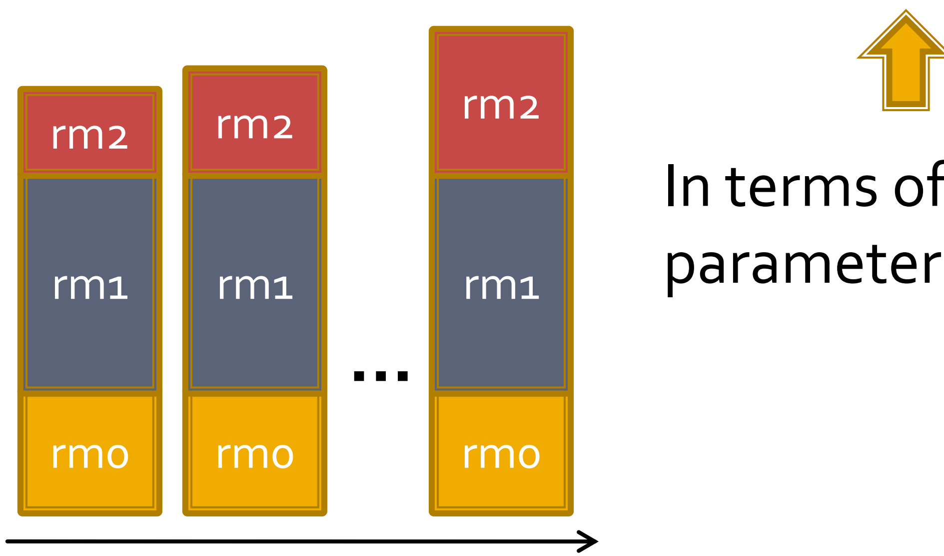
rsize(m2) = n (assume size(C)=1)

m0.1.m1.5.m2

{ k= mc, 1≤i ≤k, n = i}

{ k= mc = n} maximizes

maxrsize(m0.1.m1.5.m2,m0) = **mc**

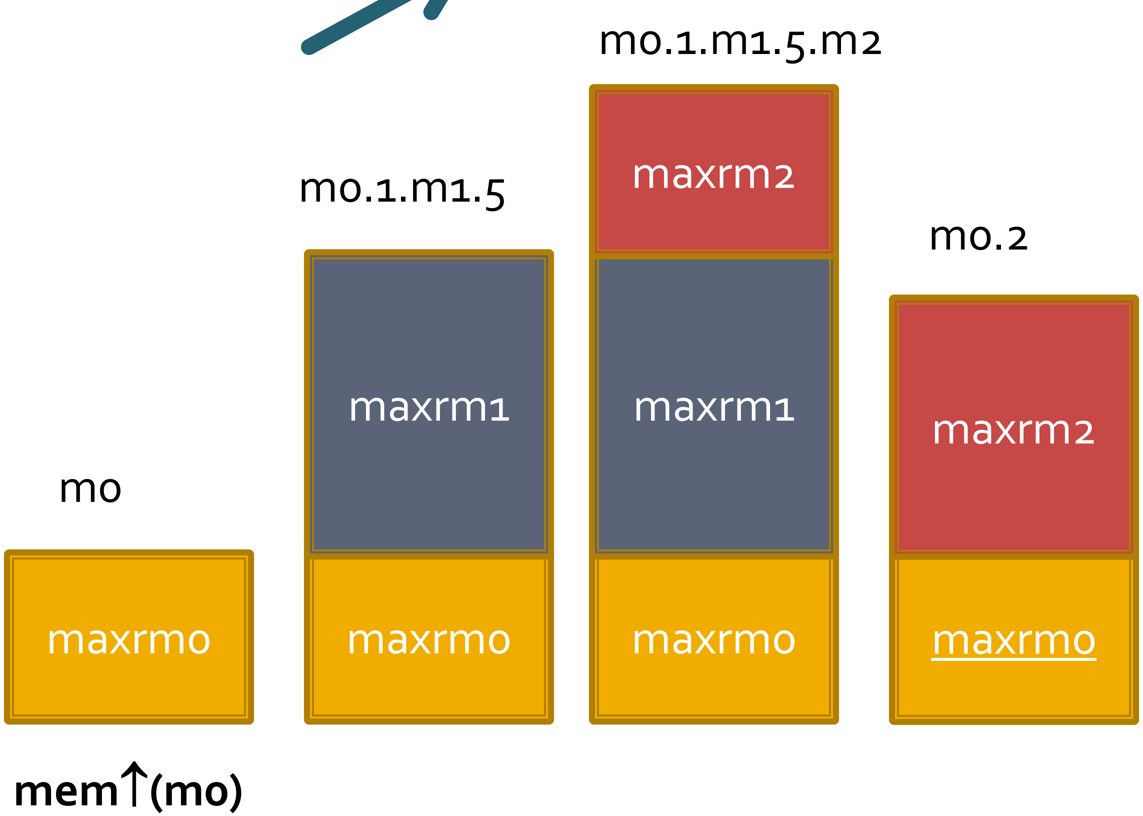


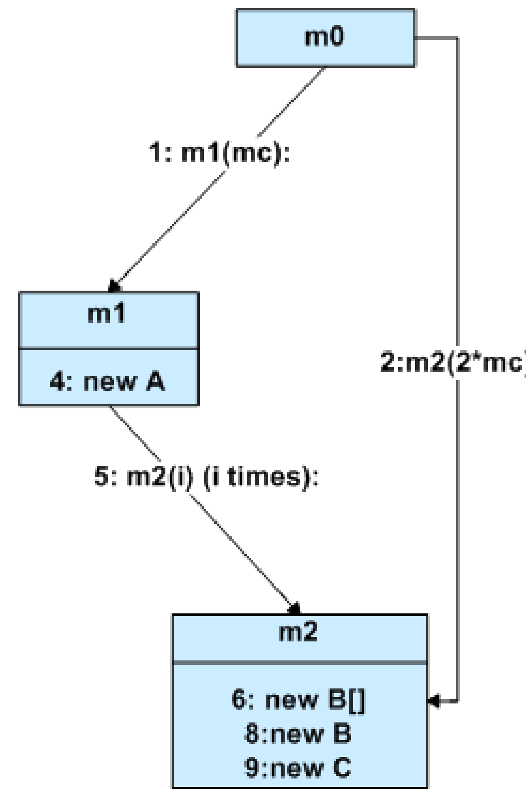
In terms of **m0** parameters!

Approximating Peak↑(m)

We consider the largest region for the same calling context

$$\text{peak}\uparrow(\text{mo}) \;\le\; \max_{\pi\in\prod_{mo}} \sum_{1\le k\le|\pi|} \text{maxrsize}_{m0}^{\pi[1..k]}(mc) \;=\; \text{mem}\uparrow(\text{mo})$$





Approximating Peak↑(m)

3. Maximizing instantiated regions

$$\text{maxrsize}(\pi.m,mo)(P_{mo}) = \text{Maximize } \textbf{\textit{rsize(m)}} \text{ subject to } \boldsymbol{I_{\pi}(P_{mo}, P_{m}, W)}$$

- m-
  - r
- Ma
  of
  - n
  - n

- We cannot solve a non-linear maximization problem in runtime!!
  - Too expensive
  - Execution time difficult to predict
- We need a parametric solution that can be solved at compile time.

Solving maxrsize

- Solution: use an approach based on Bernstein basis over polyhedral domains (Clauss et al. 2004)
  - Enables bounding a polynomial over a parametric domain given as a set of linear restraints
  - Obtains a parametric solution
- Bernstein(pol, I):
  - Input: a polynomial *pol* and a set of linear (parametric) constrains *I*
  - Return **a set** of polynomials (candidates)
    - Bound the maximum value of *pol* in the domain given by *I*

Solving maxrsize using bernstein

Example:

- Input Polynomial
  - $Q(n)=n^2-1$,
- Restriction: A parametric domain (linear restraint)
  - $D(P1,P2) = \{(i, n) \mid 1 \le i \le P1 + P2,\ i \le 3P2,\ n=i\}$
- Bernstein(Q, D) =

$$\begin{cases} D_1 = \{P1 \le 2P2\} & C_1: \{(P1+P2)^2-1, P2+P1\} \\ D_2 = \{2P2 \le P1\} & C_2: \{9P2^2-1\} \end{cases}$$

- Partial solution to our problem
  - We still need to determine symbolically maximum between polynomials
  - In the worst case we can leave it for run-time evaluation (cost known "a priori")
    - A comparison when actual parameters are available

maxrsize

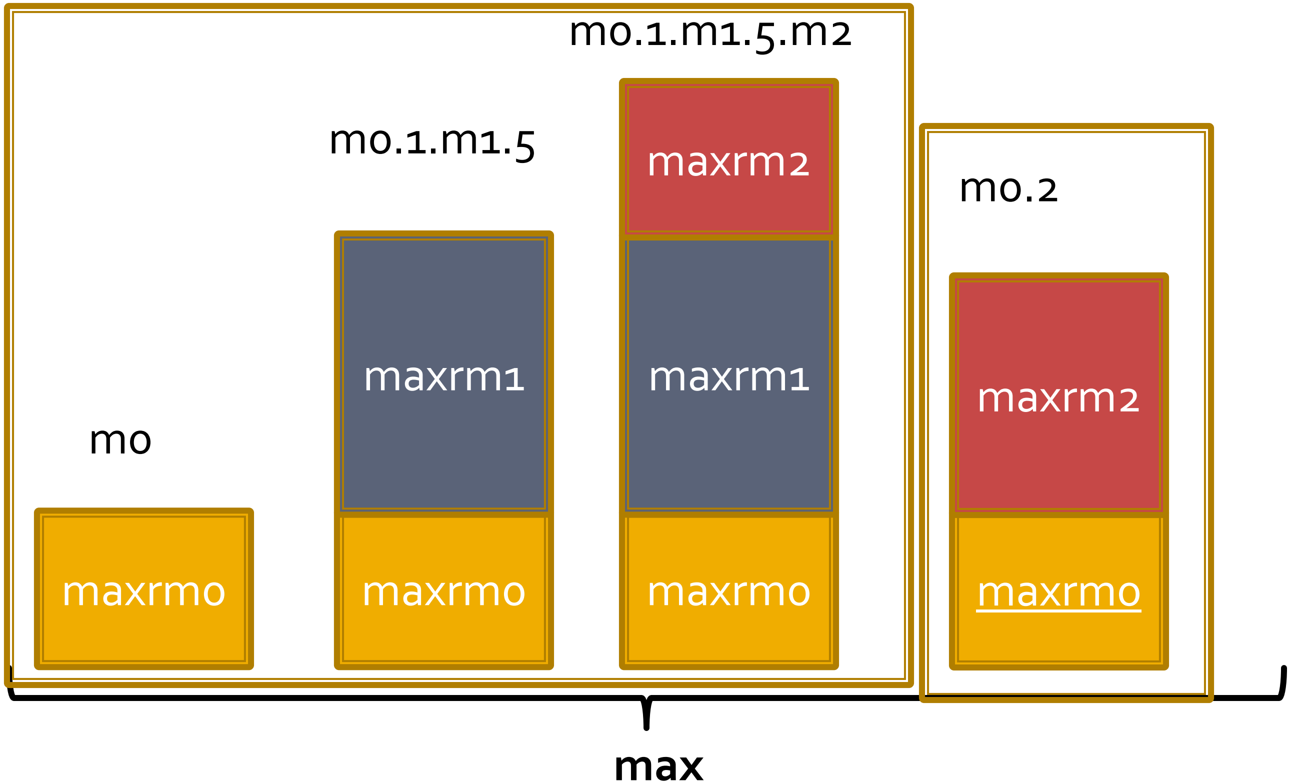
$$\text{Maxrsize}(mo,\pi.m_k) = \begin{cases} \max \{ q(P_{mo}) \in C1\} \text{ if } D1(P_{mo}) \\ \max \{ q(P_{mo}) \in Ck\} \text{ if } Dk(P_{mo}) \end{cases}$$

where $\{Ci, Di\}$ = Bernstein(rsize(m_k), $I_{\pi.mk}$, P_{mo})

- Maxrsize(mo,mo)(mc) = **(size(B[]) + size(B)).2mc**
- Maxrsize(mo.1.m1,mo) (mc) =
  (size(B[]) + size(B)).(1/2 mc² +1/2mc) +size(A).mc
- Maxrsize(mo.1.m1.5.m2,mo)(mc) = **size(C).mc**
- Maxrsize(mo.21m2,mo)(mc) = **size(C).2mc**

Evaluating mem↑

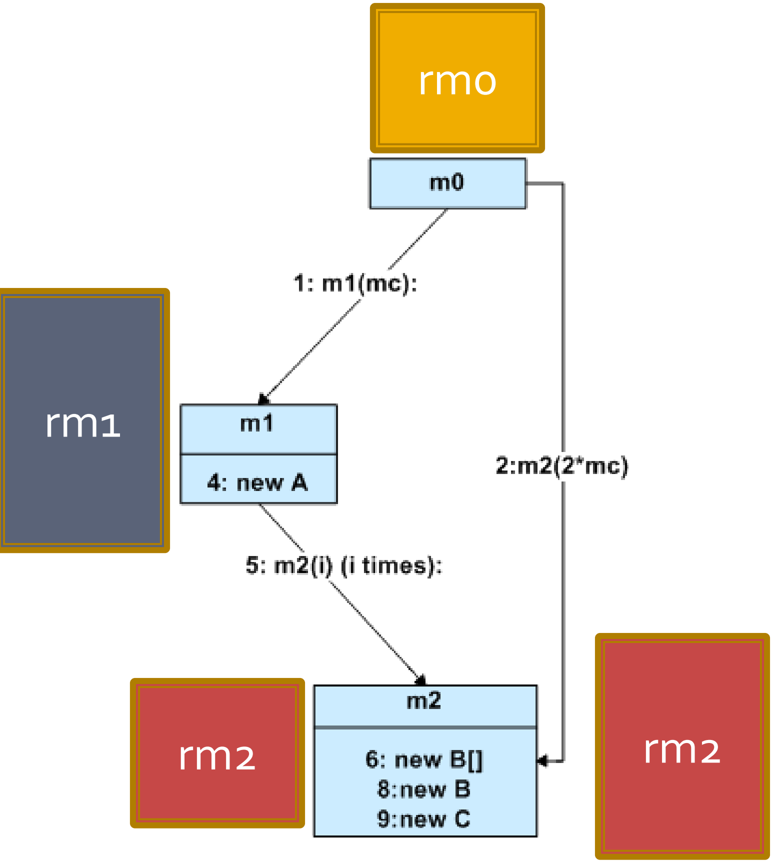
- **mem↑:** $\max_{\pi \in \prod_{mo}} \sum_{1 \le k \le |\pi|} \text{maxrsize}_{m0}^{\pi[1..k]}(mc)$
  - basically a sum maximized regions
  - A comparison of the results of the sum

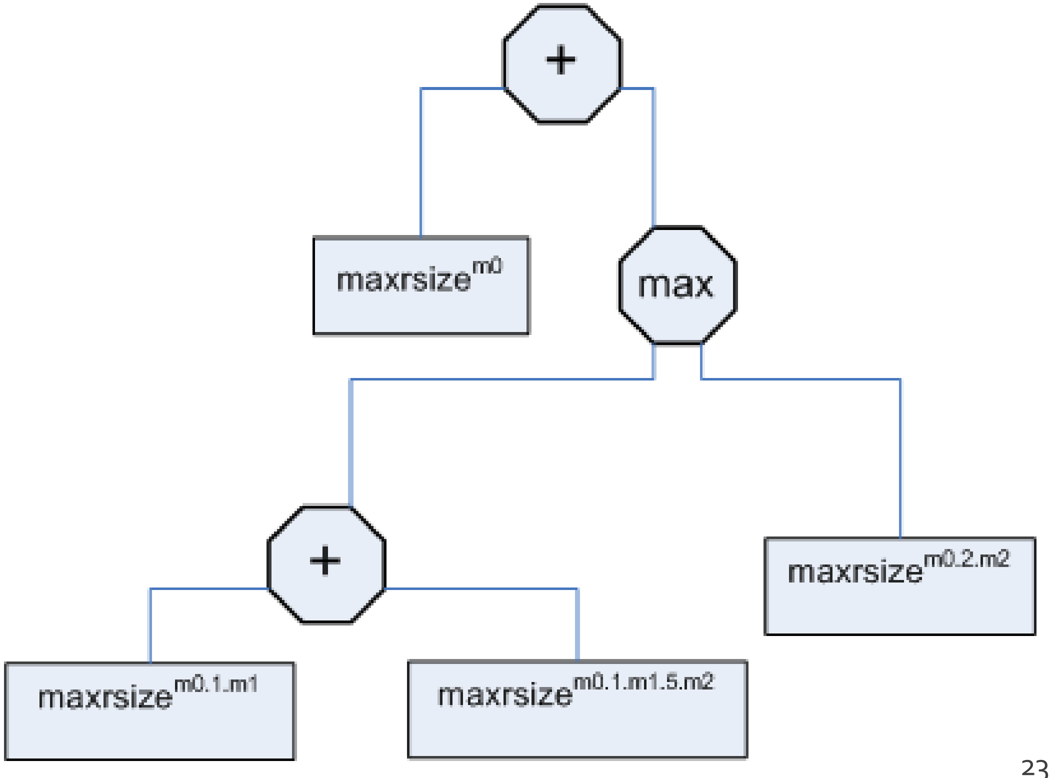


Evaluating mem↑

$$\mathrm{mem}{\uparrow}_{\mathsf{mua}} = \mathrm{mem}{\uparrow}^{\mathsf{mua}}_{\mathsf{mua}}$$

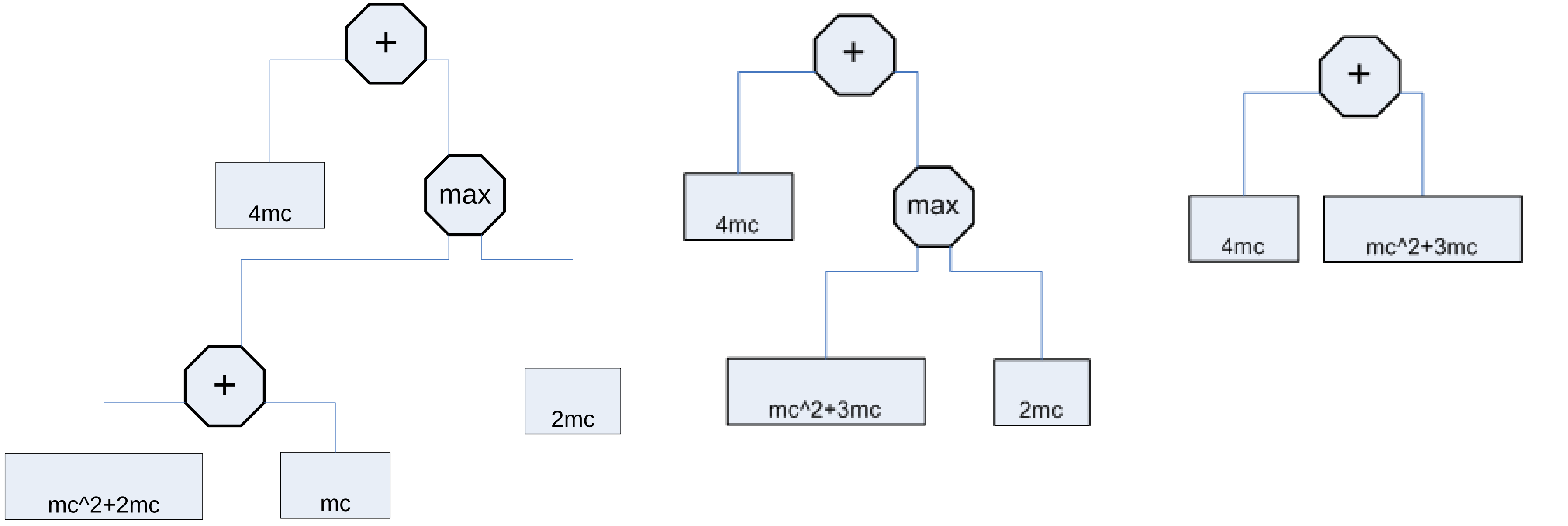
$$\mathrm{mem}{\uparrow}^{\mathsf{cst}.m}_{\mathsf{mua}} = \mathrm{maxrsize}^{\mathsf{cst}.m}_{\mathsf{mua}} + \max_{(m,l,m_i)\in\mathcal{E}_{\mathsf{mua}}} \mathrm{mem}{\uparrow}^{\mathsf{cst}.m.l.m_i}_{\mathsf{mua}}$$





Manupilating evaluation trees

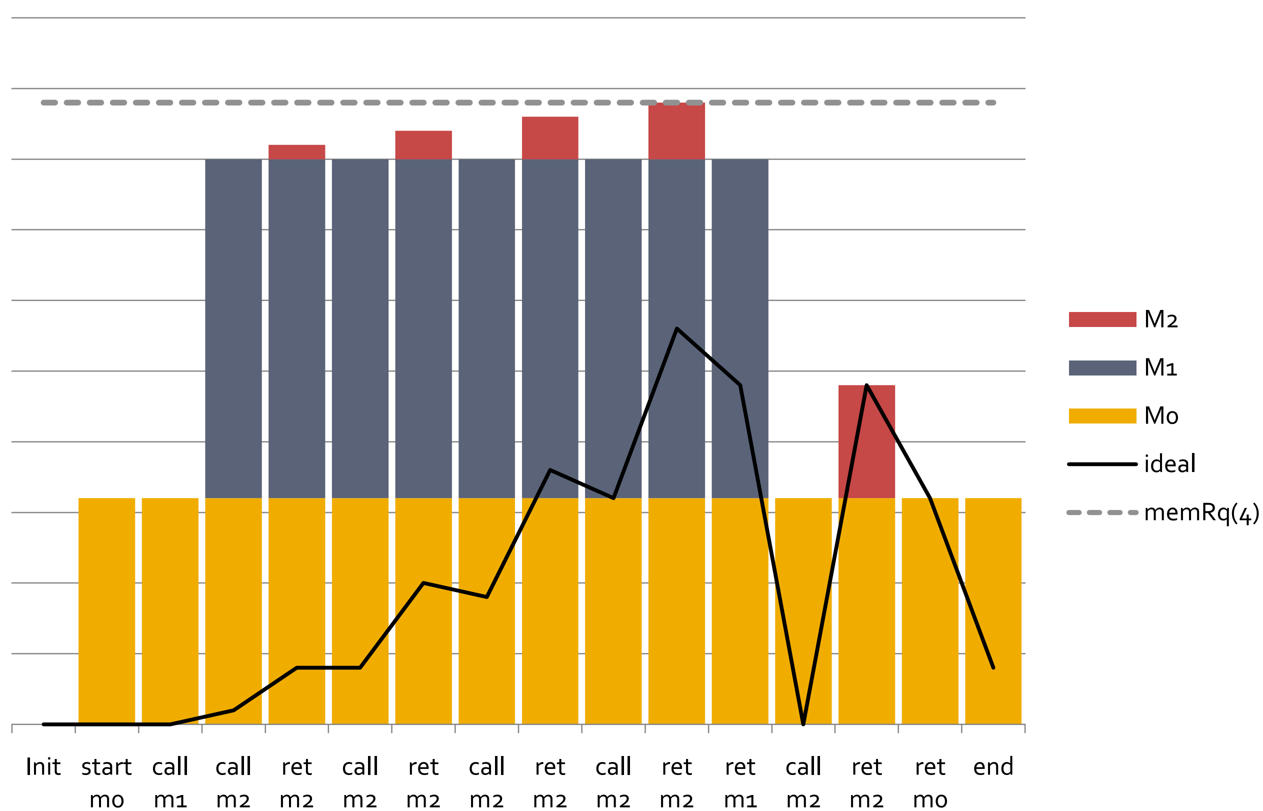
Considering size(T) = 1 for all T



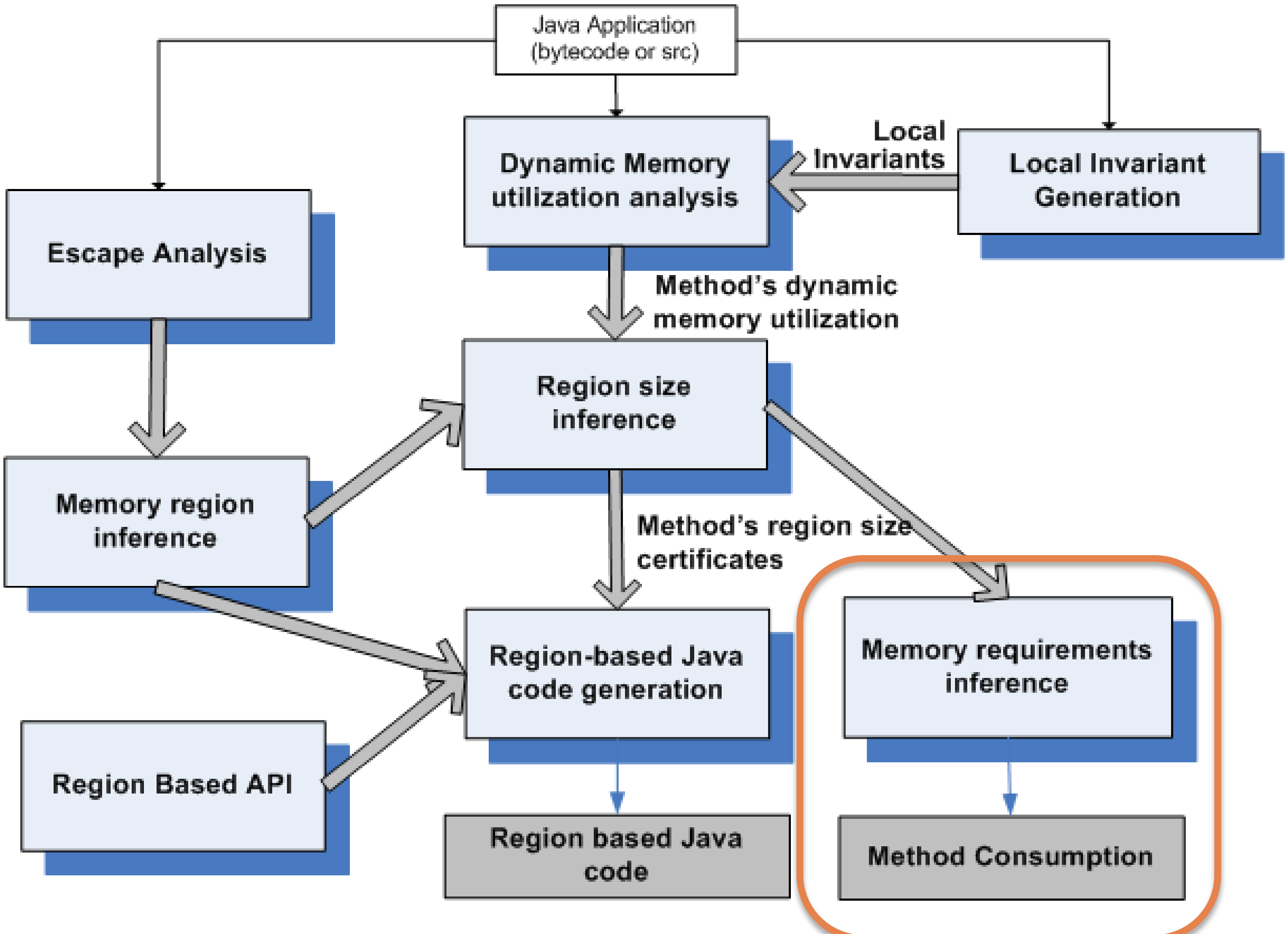
Dynamic memory required to run a method

Computing memReq

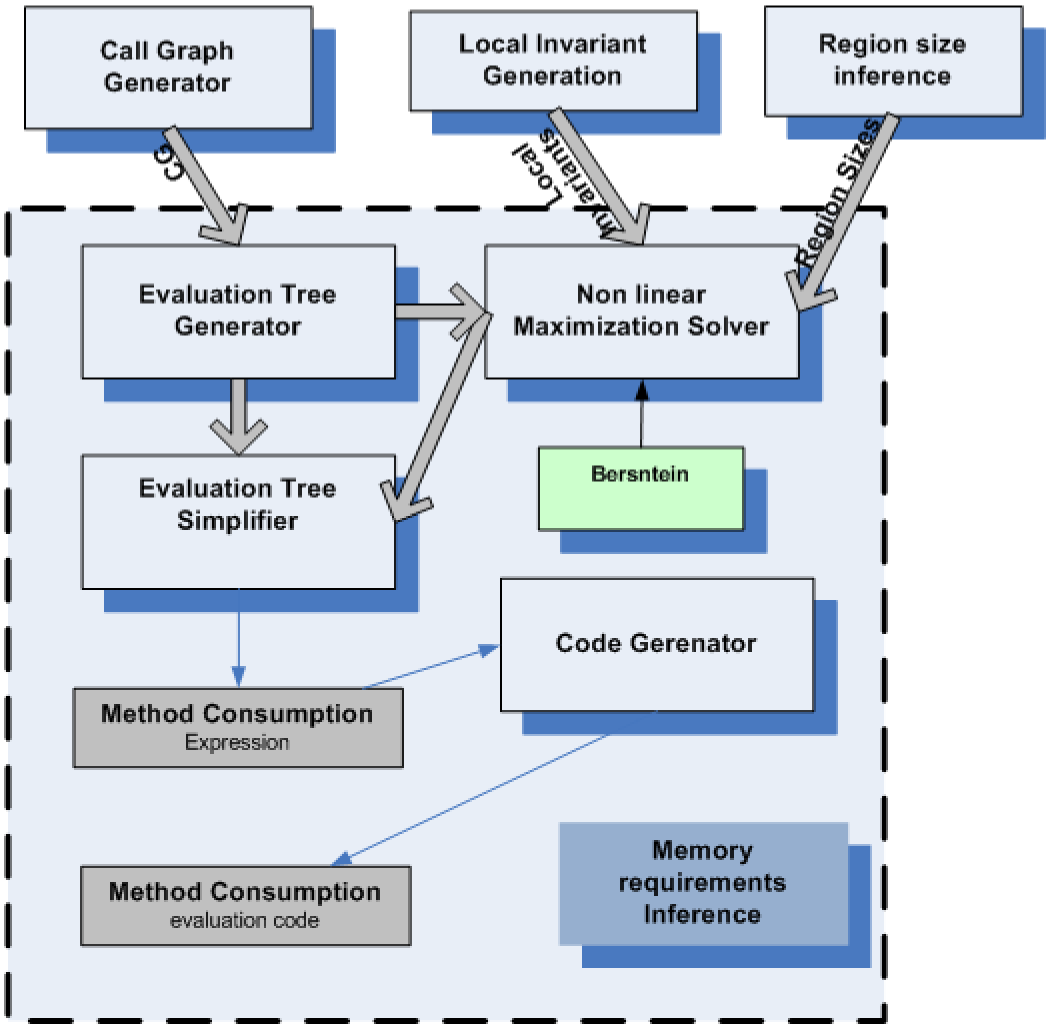
- $Memreq_{mo}(mc) = \mathbf{mc^2 + 7mc}$



The tool-suite



Peak memory computation component



Experimentation (#objects)

- Jolden: totAlloc vs Peak vs region based code

- MST, Em3d completely automatic
- For the rest we need to provide some region sizes manually
- MST, Em3d, Bisort, TSP: peak close to totAlloc (few regions, long lived objects)

•Power, health, BH, Perimeter: peak << totAlloc

Experiments (#objects)

App	$mem\uparrow_{main} + mem\downarrow_{main}$	No GC	#Regs	Param.	#Objs	Estimation	Err%	Time (secs)		
								TR	TM	TB
MST -v nv	$\frac{9}{4}nv^2+3nv+5+\max\{nv-1,2\}$	$\frac{9}{4}nv^2+4nv+6$	3	10 100 1000	253 22703 2252003	269 22904 2254004	7% 1% 0%	16.03	26.04	0.03
Em3d -n n -d d	$6n.d+2n+14+\max\{6,2n\}$	$6n.d+4n+20$	3	(10,5) (100,7) (1000,8)	344 4604 52004	354 4614 52014	3% 0% 0%	17.34	30.37	0.05
BiSort -s s -p-m	$s+4$	$2s+5$	4	10 64 128	12 68 132	14 70 134	17% 3% 3%	17.55	3.21	0.03
TSP -c c -p-m	$2x+2($ where $x=2^{(\lfloor \log_2 c\rfloor)+1})$	$4x+2$	5	10 31 63	31 63 127	34 66 130	10% 5% 2%	14.37	4.17	0.08
Power-p-m	32424	1552434	3	-	32421	32424	0%	20.72	5.82	0.02
Health -l -t	$\frac{1}{9}(21+51x+5(x-1)t)$ (where $x=4^l$)	$\frac{2}{3}(8(x-1)+(5x-2)t)$	6	(4,1) (5,3) (6,4)	1595 7510 34588	1538 7080 32791	4% 6% 5%	27.55	-	0.10
TreeAdd -l l -p-m	$x+6$ (where $x=2^l$)	$x+8$	2	8 10 12	262 1030 4102	259 1027 4099	1% 0% 0%	15.32	-	0.00
BH -b nb -s s	$13nb^2+246nb+37$	$25s.nb^2+nb(17+74s)+11s+37$	14	10 50 100	2385 11657 156637	3797 44837 23315	59% 285% 563%	25.49	-	0.08
Perimeter -l l -p-m	$x+11$ (where $x=4^{(l-4)}$)	$x+11$	2	13 14 17	158042 224090 6305002	262155 1048587 67108875	66% 367% 964%	18.78	-	0.00
Voronoi	∞	∞	5					27.76	-	-

Related work

Author	Year	Language	Expressions	Memory Manager	Benchmarks
Hofmann & Jost	2003	Functional	Linear	Explicit	No
Lui & Unnikrishnan	2003	Functional	Recursive functions	Ref. Counting	Add-hoc (Lists)
Chin et al	2005	Java like	Linear (Pressburger): Checking	Explicit	Jolden
Chin et al **NEXT PRESENTATION!**	2008	Bytecode	Linear: Inference	Explicit	SciMark, MyBench
Albert et al (2)	2007	Bytecode	Recurrence equations	No && Esc Analysis	No

Conclusions

- A technique for computing parametric (easy to evaluate) specifications of heap memory requirements
  - Consider memory reclaiming
  - Use memory regions to approximate GC
  - A model of peak memory under a scoped-based region memory manager
  - An application of Bernstein to solve a non-linear maximization problem
  - A tool that integrates this technique in tool suite
- Precision relies on several factors:
  - invariants, region sizes, program structure, Bernstein

Conclusions

Future work

- Restrictions on the input
  - Better support for recursion
  - More complex data structures
  - Other memory management mechanisms
- Usability / Scalability
  - Integration with other tools/ techniques
    - JML / Spec# (checking+inferring)
    - Type Systems (Chin et al.)
    - Modularity
  - Improve precision