A type system for analyzing the complexity of Object Oriented programs

Romain Péchoux
(joint work with Emmanuel Hainry)

Université de Lorraine, LORIA, Nancy, France

University of Dundee

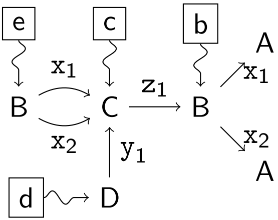
OutOfMemoryError and StackOverflowError

In Java,

- `OutOfMemoryError` is "thrown when the JVM cannot allocate an object because it is out of memory", that is when the *heap* is full.
- `StackOverflowError` is "thrown when a stack overflow occurs because an application recurses too deeply."

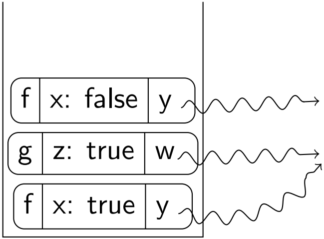
Heap

- Where objects are created and kept in memory.
- Maximal heap space is defined at the launch of the JVM.
- Pointers to the objects, arrows between objects and their attributes.



Stack

- Where arguments of a method call are put.
- Primitive types are put by value.
- Object types are put by reference, *i.e.* a pointer to the heap.
- May grow indefinitely because of recursive calls.



Objectives

Practical motivations:

- Bound the memory (Heap and Stack) Usage
  - using a polynomial algorithm;
  - in an object oriented language;
  - with advanced OO features (inheritance, recursion)

Theoretical motivations:

- Characterize well known complexity classes
  - FPtime,
  - FPspace, ...

Non exhaustive state of the art

On imperative programs:

- Matrix calculus (Ben Amram, Jones & Kristiansen, Moyen)
- Graph language (Hofmann & Schoepp)

On Object Oriented Languages:

- Amortised analysis for linear heap (Hofmann & Jost)
- "Costa" for analyzing Java bytecode (Albert, Arenas, Genaim, Puebla & Zanardini)
- "Speed" for C++ (Gulwani *et al.*)
- "ResAna" analyzes Java programs (Shkaravska *et al.*)

Tiered based "type systems" for resource analysis

- Bellantoni & Cook 1992
  - Functional setting
  - Two kinds of arguments: Safe and Normal
  - Characterizing FPTIME
- Leivant & Marion 1993
  - λ-calculus
  - n tiers (but 2 suffice)
  - Characterizing FPTIME
- Marion 2011
  - Imperative setting
  - 4 sorts ((α, β) with $\alpha, \beta \in \{0, 1\}$)
  - Characterizing FPTIME under Termination assumption

Tiers for imperative languages revisited

Expressions, variables, instructions are given a tier in $\{\mathbf{0}, \mathbf{1}\}$.

- Expressions:
  - $op(\mathtt{y})$ may be of tier $\mathbf{1}$ if $\forall \mathtt{x},\ \#\{op^n(\mathtt{x}) \mid \forall n \in \mathbb{N}\} \leq P(|x|)$.
  - $op(\mathtt{y})$ may be of tier $\mathbf{0}$ if $\forall \mathtt{x},\ |[\![op]\!](\mathtt{x})| \leq |x| + k,\ k \in \mathbb{N}$.
- Assignation
  - $X^\alpha := e^\beta : \alpha$ provided that $\alpha \leq \beta$.
  - Non-interference like typing rule (flows from $\mathbf{1}$ to $\mathbf{0}$ only).
- Conditional
  - if e^α then I_1: α else I_2: α
- Loop
  - While e^1 do I: α

If a *terminating* program can be tiered, it is in FPTIME.

Example: addition

```
int add(int x,int y)
{
while (x>0)
        {
        x--;
        y++;
        }
return y
}
```

- y is necessarily of tier $\mathbf{0}$
- x is necessarily of tier $\mathbf{1}$
- and, consequently, add $:: \mathbf{1} \times \mathbf{0} \rightarrow \mathbf{0}$

Example: multiplication

```
int mult(int x,int y)
{
int z=0;
while (x>0)
      {
      x--;
      z = add(y,z);
      }
return z;
}
```

- the output of add is $\mathbf{0}$. Consequently, z is of tier $\mathbf{0}$.
- both x and y are of tier $\mathbf{1}$
- and, consequently, `mult` $:: \mathbf{1} \times \mathbf{1} \to \mathbf{0}$

Example: exponential

```
int expo(int x)
{
int y=1;
while (x>0)
          {
          x--;
          y = add(y,y);
          }
return y;
}
```

- x is of tier $\mathbf{1}$,
- the output of add is of tier $\mathbf{0}$,
- but y has to be of tier $\mathbf{1}$ in the first argument of `add` !!!

Core Java

- Expressions
  $$E ::= \; \mathtt{x} \mid \mathtt{null} \mid \mathtt{this} \mid n \mid \mathtt{true} \mid \mathtt{false} \mid op(\overline{E}) \mid \mathtt{new}\ \mathsf{C}(\overline{E}) \mid E.m(\overline{E})$$
- Instructions
  $$I ::= \; ; \mid [\tau]\ \mathtt{x}{:=}E; \mid I_1\ I_2 \mid \mathtt{while}(E)\{I\} \mid \mathtt{x}{+}{+}; \mid \mathtt{x}{-}{-}; \mid \mathtt{break}; \mid \mathtt{if}(E)\{I_1\}\mathtt{else}\{I_2\} \mid E.m(\overline{E});$$
- Methods
  $$M_{\mathsf{C}} ::= \tau\ m(\tau_1\ \mathtt{x}_1, \dots, \tau_n\ \mathtt{x}_n)\{I[\mathtt{return}\ \mathtt{x};]\}$$
- Constructors
  $$K_{\mathsf{C}} ::= \mathsf{C}(\tau_1\ \mathtt{y}_1, \dots, \tau_n\ \mathtt{y}_n)\{\mathtt{x}_1{:=}\mathtt{y}_1; \dots \mathtt{x}_n{:=}\mathtt{y}_n;\}$$
- Classes
  $$\mathfrak{C} ::= \mathsf{D}\ \mathtt{extends}\ \mathsf{C}\{\tau_1\ \mathtt{x}_1; \dots; \tau_n\ \mathtt{x}_n;\ K_{\mathsf{C}}\ M^1_{\mathsf{C}} \dots M^k_{\mathsf{C}}\}$$

Core Java Programs

Definition [Core Java Program]

A Core Java Program is a collection of classes and exactly one executable:

$$\text{Exe}\{\texttt{main}()\{\underbrace{\tau_1\ \texttt{x}_1 := E_1; \ldots ; \tau_n\ \texttt{x}_n := E_n;}_{\text{Initialization}}\ \underbrace{I}_{\textit{Computation}}\ \}\}.$$

```
Exe {
    main() {
        boolean x = true;
        BList b1 = new BList(x, null);
        BList b2 = new BList(false, b1);
        // End of initialization
        while (true) {
            b2 = new BList(false, b2);
        }
    }
}

BList {
    boolean value;
    BList queue;

    BList(boolean v, BList q) {
        value = v;
        queue = q;
    }
}
```

Tiered types

- Expressions, Instructions, Constructors and Methods are annotated by tiered types $\tau(\alpha)$ (*i.e.* a type τ and a tier α).
- For instructions, the tier types will always be $\texttt{void}(\alpha)$.
- For methods, the tiered type is functional and the of the caller object tiered type is included:
  e.g. for `void setQueue(BList q) {...}`

$$\texttt{BList}(0) \times \texttt{BList}(1) \rightarrow \texttt{void}(0)$$

Typing Expressions

$$\frac{w \in \{\texttt{true}, \texttt{false}\}}{\Gamma \vdash w : \texttt{boolean}(\alpha)}\ \textit{(True/False)} \qquad \frac{n :: \texttt{int}}{\Gamma \vdash n : \texttt{int}(\alpha)}\ \textit{(Cst)}$$

$$\frac{\Gamma \vdash \texttt{x} : \texttt{int}(\alpha)}{\Gamma \vdash \texttt{x--} : \texttt{void}(\alpha)}\ \textit{(Dec)} \qquad \frac{\Gamma \vdash \texttt{x} : \texttt{int}(\mathbf{0})}{\Gamma \vdash \texttt{x--} : \texttt{void}(\mathbf{0})}\ \textit{(Inc)}$$

$$\frac{\alpha \preceq \min\{\text{tiers of the attributes}\}}{(m^{\mathsf{C}}, \Delta) \vdash \texttt{this} : \mathsf{C}(\alpha)}\ \textit{(Self)} \qquad \frac{\Delta(m^{\mathsf{C}})(\texttt{x}) = \tau(\alpha)}{(m^{\mathsf{C}}, \Delta) \vdash \texttt{x} : \tau(\alpha)}\ \textit{(Var)}$$

$$\frac{\forall i\ \Gamma \vdash E_i : \tau_i(\alpha) \qquad op :: \tau_1 \times \cdots \times \tau_n \to \texttt{boolean}}{\Gamma \vdash op(E_1, \ldots, E_n) : \texttt{boolean}(\alpha)}\ \textit{(Op)}$$

Typing Instructions

$$\frac{}{\Gamma \vdash ;\ : \texttt{void}(\mathbf{0})}\ \textit{(Skip)}$$

$$\frac{\Gamma \vdash \texttt{x} : \tau(\alpha) \qquad \Gamma \vdash E : \tau(\beta) \quad \alpha \preceq \beta}{\Gamma \vdash [\tau]\ \texttt{x:=}E; : \texttt{void}(\alpha)}\ \textit{(Ass)}$$

$$\frac{\Gamma \vdash I : \texttt{void}(\alpha) \quad \alpha \preceq \beta}{\Gamma \vdash I : \texttt{void}(\beta)}\ \textit{(Sub)}$$

$$\frac{\forall i\ \Gamma \vdash I_i : \texttt{void}(\alpha_i)}{\Gamma \vdash I_1\ I_2 : \texttt{void}(\alpha_1 \vee \alpha_2)}\ \textit{(Seq)}$$

$$\frac{\Gamma \vdash E : \texttt{boolean}(\alpha) \quad \forall i\ \Gamma \vdash I_i : \texttt{void}(\alpha)}{\Gamma \vdash \texttt{if}(E)\{I_1\}\texttt{else}\{I_2\} : \texttt{void}(\alpha)}\ \textit{(If)}$$

$$\frac{\Gamma \vdash E : \texttt{boolean}(\mathbf{1}) \quad \Gamma \vdash I : \texttt{void}(\mathbf{1})}{\Gamma \vdash \texttt{while}(E)\{I\} : \texttt{void}(\mathbf{1})}\ \textit{(Wh)}$$

Typing Constructors

Consider a constructor of the shape:

$$\mathsf{C}(\ldots \tau_i\ \mathtt{y}_i \ldots)\{\ldots \mathtt{x}_i := \mathtt{y}_i; \ldots\}$$

$$\frac{\forall i\ (m^{\mathsf{C}}, \Delta) \vdash E_i : \tau_i(\alpha_i) \quad (\epsilon, \Delta) \vdash \mathtt{y}_i : \tau_i(\alpha_i)}{(m^{\mathsf{C}}, \Delta) \vdash \mathtt{new}\ \mathsf{C}(E_1, \ldots, E_n) : \mathsf{C}(\mathbf{0})}\ \textit{(New)}$$

Constructors make the heap increase, hence output something of tier $\mathbf{0}$.

Typing Methods

Given a method m of the class C of the shape:

$$\tau\ m(\ldots, \tau_i\ \mathtt{x}_i, \ldots)\{I\ \mathtt{return}\ \mathtt{x};\}$$

$$\frac{\forall i\ (m_1^{\mathsf{C}_1}, \Delta) \vdash E_i : \tau_i(\alpha_i) \quad (m_1^{\mathsf{C}_1}, \Delta) \vdash E : \mathsf{C}(\beta) \\ (m^{\mathsf{C}}, \Delta) \vdash m : \mathsf{C}(\beta) \times \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \tau(\alpha)}{(m_1^{\mathsf{C}_1}, \Delta) \vdash E.m(E_1, \ldots, E_n) : \tau(\alpha)} \ (Call)$$

$$\frac{(m^{\mathsf{C}}, \Delta) \vdash \mathtt{this} : \mathsf{C}(\beta) \qquad \forall i\ (m^{\mathsf{C}}, \Delta) \vdash \mathtt{x}_i : \tau_i(\alpha_i) \\ (m^{\mathsf{C}}, \Delta) \vdash \mathtt{x} : \tau(\alpha) \qquad (m^{\mathsf{C}}, \Delta) \vdash I : \mathtt{void}(\alpha)}{(\epsilon, \Delta) \vdash m : \mathsf{C}(\beta) \times \cdots \times \tau_i(\alpha_i) \times \cdots \to \tau(\alpha)} \ (M_C)$$

The tier of the output is that of the returned value and of the instruction (modulo subtyping).
Note that the tier of `this` must be known for tiering the method.

Example

Concatenation is typable:

```
void concat(BList other){
  BList o = this;
  while (o.getQueue() != null){
     o = o.getQueue();
   }
  o.setQueue(other);
}
```

- other has to be of type $\mathit{BList}(\mathbf{1})$ in the setQueue call
- getQueue is of type $\mathit{BList}(\alpha) \to \mathit{BList}(\mathbf{1})$ in the while guard
- o may be of tier $\mathbf{0}$ or $\mathbf{1}$
- concat has type $\mathit{BList}(\alpha) \times \mathit{BList}(\mathbf{1}) \to \texttt{void}(\mathbf{1})$

Example

List generation is typable:

```
void generate(int n){
  BList o = null;
  while (n>0){
     o = new BList(true,o);
     n--;
   }
  return o;
}
```

- n has type $\texttt{int}(\mathbf{1})$ because of the while guard
- o has tier $\mathbf{0}$ because of the new
- $n--$ is typable
- generate has type $\mathsf{C}(\alpha) \times \texttt{int}(\mathbf{1}) \rightarrow \mathit{BList}(\mathbf{0})$

Example

Creation is typable:

```
void concat(BList other){
  BList o = this;
  while (o.getQueue() != null){
     o = o.getQueue();
   }
  o.setQueue(other);
}
```

- other has to be of type $BList(\mathbf{1})$ in the setQueue call
- getQueue is of type $BList(\alpha) \rightarrow BList(\mathbf{1})$ in the while guard
- o may be of tier $\mathbf{0}$ or $\mathbf{1}$

Safety assumption

Definition [Safety]

A well-typed program with respect to a typing environment Δ is *safe* if for each recursive method $M_C = \tau\ m(\ldots)\{I\ [\texttt{return x;}]\}$:

- there is exactly one call (even nested) to m,
- there is no while loop inside I,
- and the following judgment can be derived:

$$(\epsilon, \Delta) \vdash M_C : \mathsf{C}(\mathbf{1}) \times \tau_1(\mathbf{1}) \times \cdots \times \tau_n(\mathbf{1}) \to \tau(\mathbf{1}).$$

Main result

Theorem

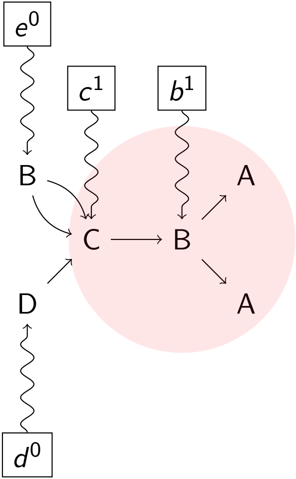
In the execution of a *safe* Core Java program *terminating* on input $\mathcal{C}$, the size of the heap and of the stack are in $O(|\mathcal{C}|^{n_{\mathbf{1}}((\nu+1)\lambda)})$.

With

- n_1 the number of variables and attributes of tier $\mathbf{1}$,
- λ the maximum number of nested while and
- ν the maximum number of nested methods.

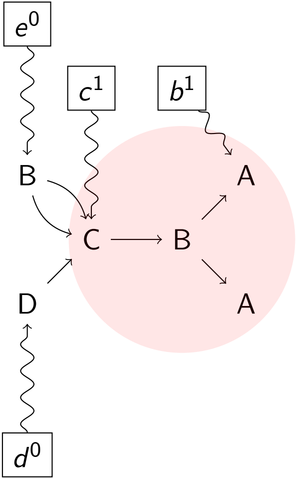
Note that $n_{\mathbf{1}}((\nu+1)\lambda)$ is a constant polynomial in the size of the program.

Idea of the proof



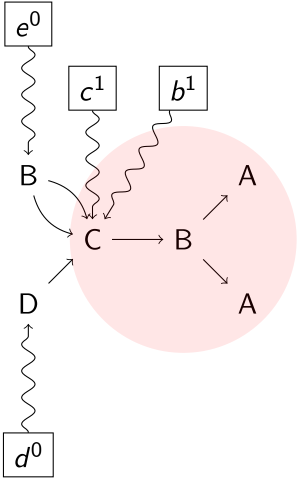
- The subheap of tier **1** never grows.
- Only tier **1** variables control `while` and recursive functions.
- The number of tier **1** configurations is bounded by $|\mathcal{C}|^{2\times n_1}$.
- Hence a bound on the stack and heap.

Idea of the proof



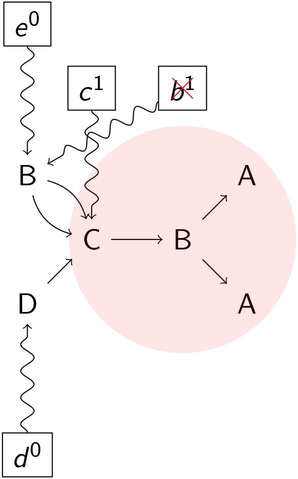
- The subheap of tier **1** never grows.
- Only tier **1** variables control `while` and recursive functions.
- The number of tier **1** configurations is bounded by $|\mathcal{C}|^{2\times n_1}$.
- Hence a bound on the stack and heap.

Idea of the proof



- The subheap of tier **1** never grows.
- Only tier **1** variables control `while` and recursive functions.
- The number of tier **1** configurations is bounded by $|\mathcal{C}|^{2\times n_1}$.
- Hence a bound on the stack and heap.

Idea of the proof



- The subheap of tier **1** never grows.
- Only tier **1** variables control `while` and recursive functions.
- The number of tier **1** configurations is bounded by $|\mathcal{C}|^{2\times n_1}$.
- Hence a bound on the stack and heap.

Type inference

Proposition [Type inference]

The type inference can be done in time polynomial in the size of the program.

Note There being no typing does not preclude the program from running in polynomial space or time.

Extensional completeness

Theorem [FPtime]

Every function computable by a TM in polynomial time can be computed by a safe, terminating and typable program.

- Soundness: every reduction is polynomial.
- Completeness: every polynomial can be computed and we write a program simulating a TM.

Conclusion

Result

- Static typing to guarantee memory bounds in OO Languages
- Explicit bounds (can be tightened)
- Expressivity:
  - recursive functions
  - inheritance and other Object Oriented features
  - control flow statements such as `break` or `continue`

Drawbacks and Open questions

- Not intentionally complete
- Obviously does not take memory leaks in the VM into account
- Thread Creation?
- Garbage Collecting?