Identification of unnecessary object allocations using static escape analysis

Faouzi Mokhefi
Stéphane Ducasse
Pablo Tesone
Luc Fabresse

CRIStAL
Centre de Recherche en Informatique, Signal et Automatique de Lille

IMT Nord Europe
École Mines-Télécom
IMT-Université de Lille

ESUG'25

International Worshop for Smalltalk Technologies 2025

Motivating Example

- OOP encourages frequent object creation —> Abundance of short-lived objects.
- The impact on memory management and garbage collector.

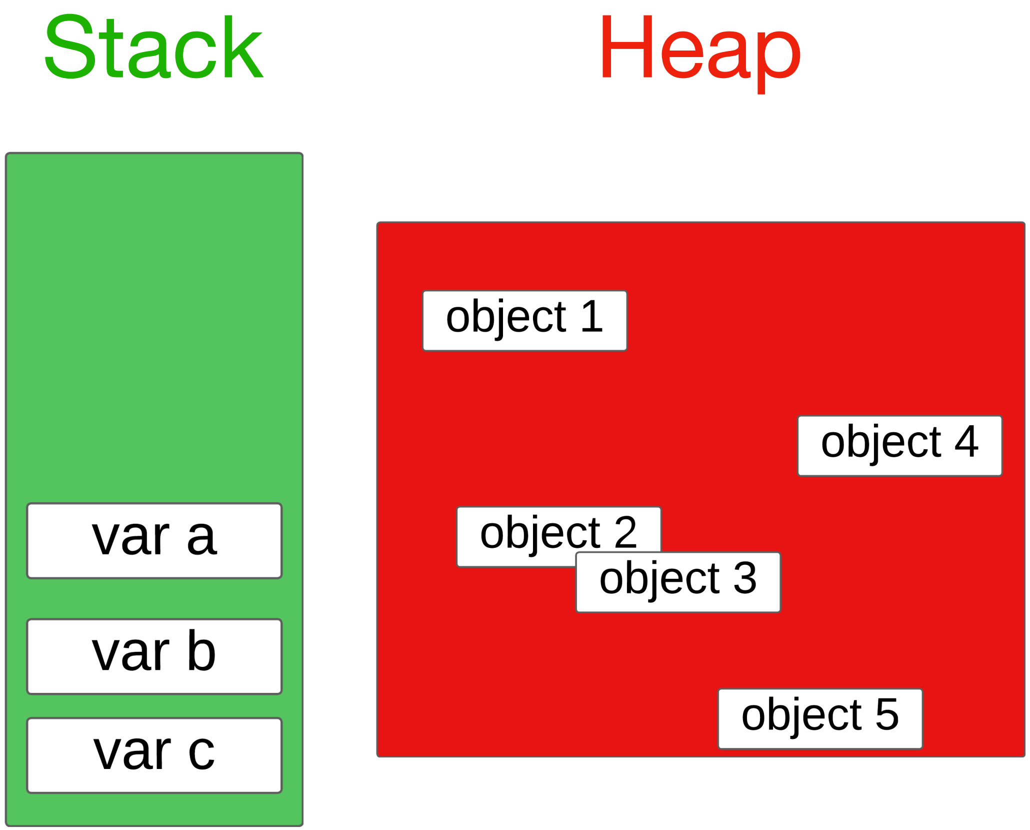
```
carFactory
    | cars res|

    cars := OrderedCollection
new.
    1 to: 100000 do: [ :i |
        cars add: (Car
            name: ‘ … ‘
            model:’ … ’
              ].
 ^ cars sum: [:aCar | aCar
price]
```

What is the impact of this code ?

Can we optimise it?

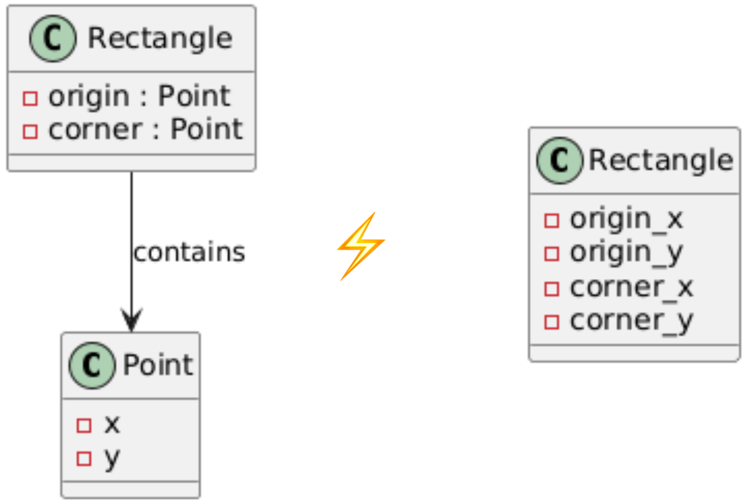
Possible Optimisations

- Stack allocation
- Object inlining
- ….



Possible Optimisations

- Stack allocation
- **Object inlining**
- ....



Rectangle origin: 1@1 corner: 2@2

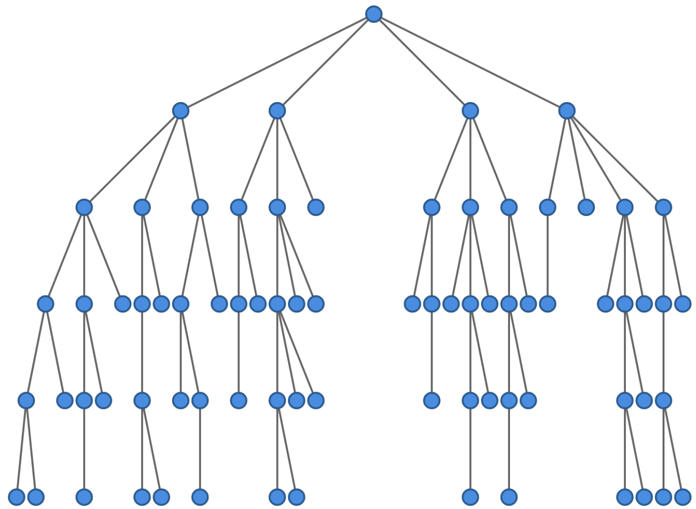
How to identify objects that can be inlined/stack allocated?

Challenges of dynamically-typed language analysis

- Dynamically-typed languages contain highly polymorphic call sites

- Prevalent use of reflection and block closure

- Large call graphs



Large call graphs

67% selectors have multiple Implementors

Characterizing escaping objects

Non-escaping object: lifetime is bound to its allocation context

methodWithNonEscaping

Mybuilder new

Build.

Escape analysis identifies, at compile-time, objects that are not reachable outside of a given execution context.

An escaping object

methodWithEscaping

```
o := Object new.
aGlobal := Array new: 7.
^ o
```

Our escape causes

- *Assignments* to *instance variables* of heap-allocated objects
- *Assignments* to *global/shared* variables
- *Return* to the top level of the call stack
- *Arguments* of blocks

Known selectors that cause escape

- Reflective, primitive, ffis e.g. #perform:with:
- Collection constructors, e.g. #with:,…
- Block evaluation e.g. #value:

Selectors to skip

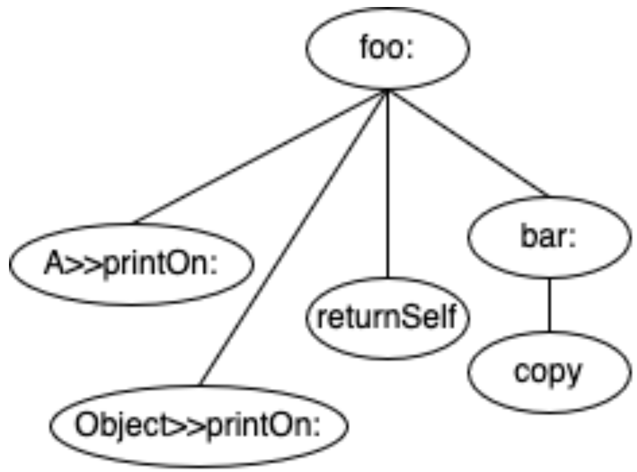
- Object creation
- Exception handling and control flow
- Testing methods
- Error messages and halts
- asString asSymbol printString

Our approach

- Context-sensitive, flow-insensitive escape analysis for Pharo
- Builds a points-to graph to track references to heap-allocated objects
- Applies escape constraints on the points-to graph
- Interprocedurality to handle method calls

Need for interprocedural analysis

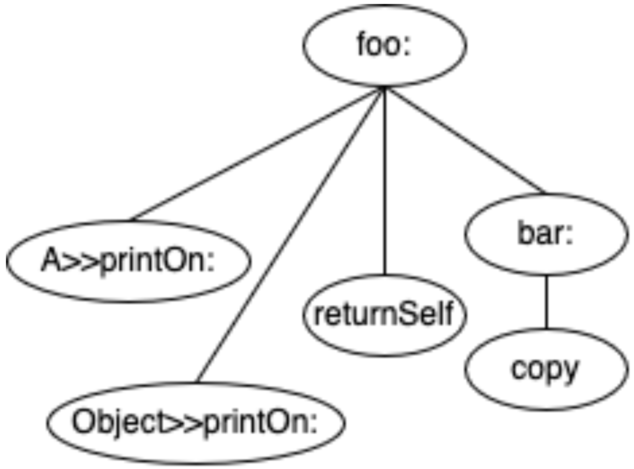
```
A >> foo: arg
    | tmp st st1 |
    tmp := Object new.
    st := Stream new.
    self bar: tmp.
    st1 := st returnSelf.
    tmp printOn: st
A >> bar: anObject
        ^ anObject
copy
```



Need for interprocedural analysis

```
A >> foo: arg
    | tmp st st1 |
    tmp := Object new.
    st := Stream new.
    self bar: tmp.
    st1 := st returnSelf.
    tmp printOn: st

A >> bar: anObject
        ^ anObject
copy
```

Call Edge

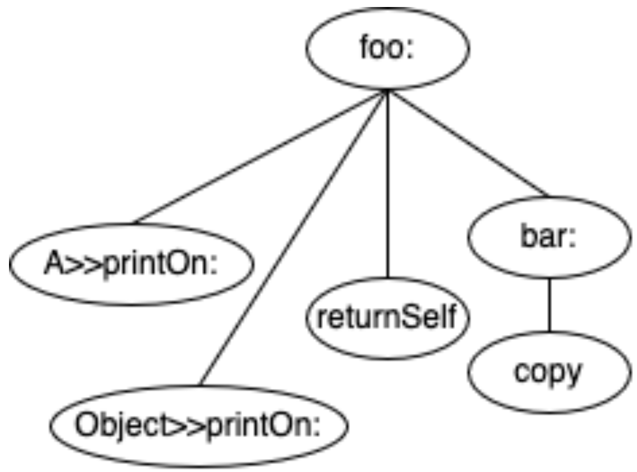


Need for interprocedural analysis

```
A >> foo: arg
    | tmp st st1 |
    tmp := Object new.
    st := Stream new.
    self bar: tmp.
    st1 := st returnSelf.
    tmp printOn: st

A >> bar: anObject
        ^ anObject
copy
```

Return Edge

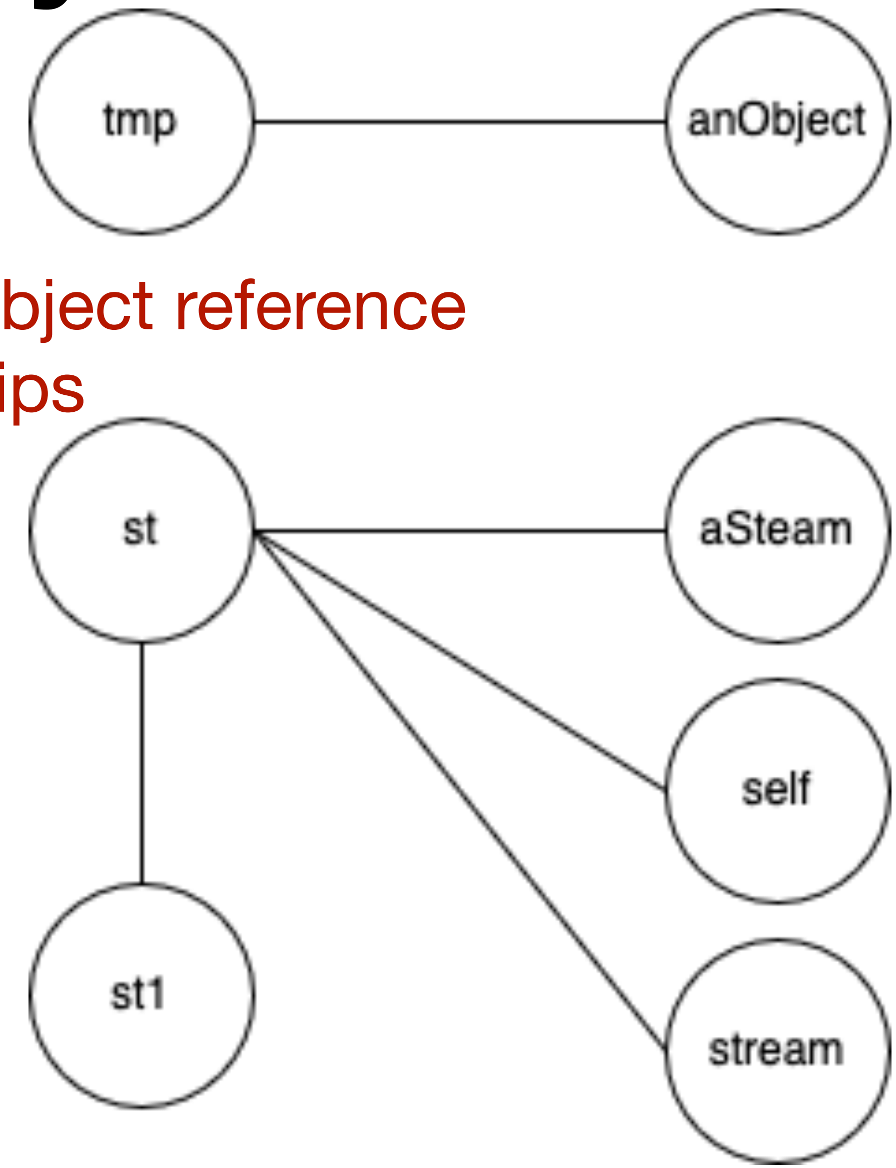
Call Edge



Points-to Analysis

```
A >> foo: arg
    | tmp st st1 |
    tmp := Object new.
    st := Stream new.
    self bar: tmp.
    st1 := st returnSelf.
    tmp printOn: st

A >> bar: anObject
        ^ anObject
copy
```

Capture object reference relationships



Addressing our challenges

- Call site memoization
- Call graph node skipping
- Type propagation
- Graph depth and breadth heuristic limitations

Call site Memoization

- Save and reuse analysis summaries for call sites based on:
  - Arguments monitoring status (tracked or not) and type information
  - Selector identity

```
A >> foo: arg
   | tmp st st1 |
   tmp := Object new.
   st := Stream new.
   self bar: tmp.
   st1 := st returnSelf.
➡  st1 bar: st.
   tmp printOn: st
A >> bar: anObject
        ^ anObject
copy
```

Call graph node skipping

When a representation of a method includes only escaping variables to track, there is no further information to extract —> Safe to skip.

Type propagation

- The analyzer models all possible implementor methods for a message

- We propagation variables types starting from allocation site and leverage to reduce scope of implementors

Call graph complexity limitation

- Open world assumption
- Limit the depth of graph exploration
- Limit Working list size (#visited methods)

Results

Anecdotal evidence

```
AIAstar >> heuristicFrom: startModel to: endModel
    | dijkstra addEdges pathD parameters |
    …
    parameters := OrderedCollection new.
    parameters add: startModel model.
    parameters add: endModel model.
    dijkstra start: parameters first.
    dijkstra end: parameters second.
    …
```

Depth impact on analysis

Depth	Analysis time	Candidates
1	9 min 14 s (39 ms)	54
2	47 min 41 s (204 ms)	189
3	3 h 29 min (895 ms)	199
4	27 h 11 min (270712995 ms)	203
≥5	> 27 h	—

Table 2: Average *SubMethods* Depth of call graph. Runtime and resulting positive candidates – fixed number of input methods.

- 203 candidates
- Depth = 3 was found to be a stable point for a reasonable execution time.

Time to perform analysis

Total methods	Analysis time	Candidates
4 800	6 min 32 s	103
7 200	12 min 32 s	138
9 600	16 min 35 s	147
12 000	19 min 46 s	158
14 400	1 h 32 min 43 s	166
16 800	35 min 57 s	173
19 200	41 min 10 s	177
21 600	46 min 48 s	182
24 000	2 h 35 min 38 seconds	189

Table 3: Runtime and resulting positive candidates - variable number of input methods for maximum depth of 2 for the call graph

How many special methods encountered

Depth	FFI	Primitives	Reflection
1	0	226	32
2	1458	15196	14582

Type Propagation Evaluation

Depth n	Analysis time w/ TP	Analysis time w/o TP	Candidates w/ TP	Candidates w/o TP
1	5 s	7 s	12	12
2	20 s	56 s	43	38
3	1 min 1 s	2 min 9 s	48	38
4	1 min 53 s	3 min 30s	49	39
5	3 min 33 s	1 h 1 min	48	37
6	7 min 50 s	18 min 09 s	48	39
7	14 min 39 s	45 min 1 s	48	37
8	48 min 40 s	—	48	—
9	10 h 1 min	—	47	—

Table 4: Analysis time and candidate counts across depths, with and without type propagation (tp: Type propagation).

False negatives:

Objects marked as escaping by static analyzer
But non-escaping during execution

- Objects returned from a constructor or factory
- Objects assigned to instance variables of non-escaping objects

Perspective

- Analyze individual fields of objects rather than treating the entire object as an indivisible unit.
- Analyse beyond Assignments to Instance Variables.
- Adding the abstract interpretation (partially evaluate some messages if we have all their argument values)
- Towards analysing external projects.

Summary

- Many short-lived objects in OOP
- Context-sensitive, flow-insensitive, interprocedural escape analysis for Pharo
- Builds an points-to graph to track references to heap-allocated objects
- Conservative escape conditions to make faster analysis
- Limited results (203 out of 24000~ allocation)