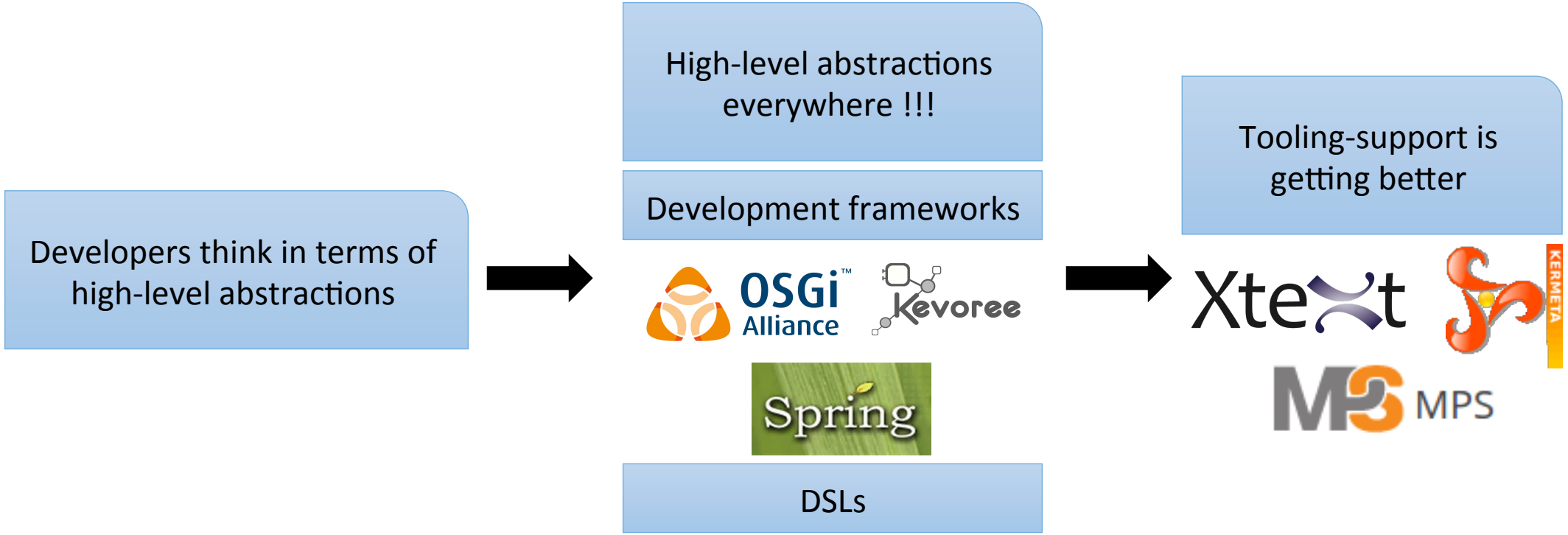
Building custom memory profilers

Inti Gonzalez-Herrera
Diverse Team
Advisers: Johann Bourcier and Olivier Barais

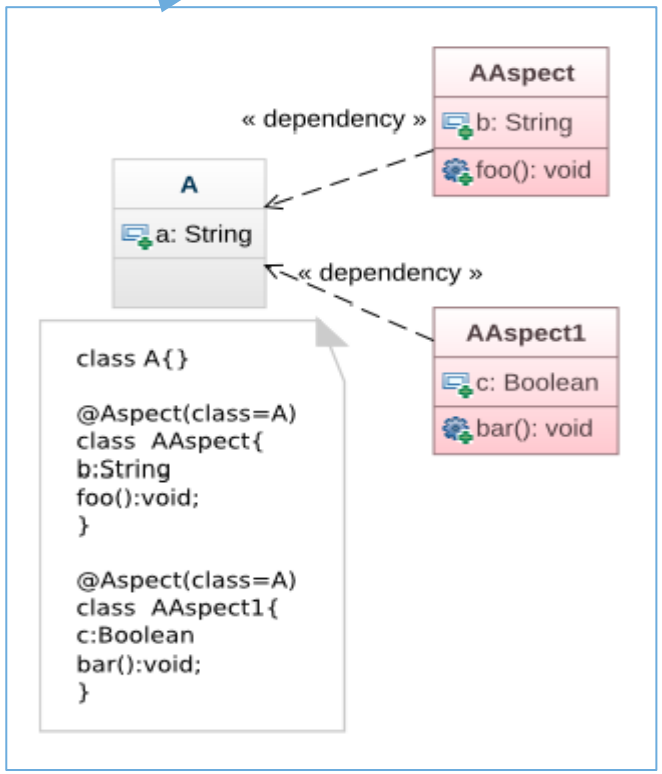
What are and why we need custom memory profilers?



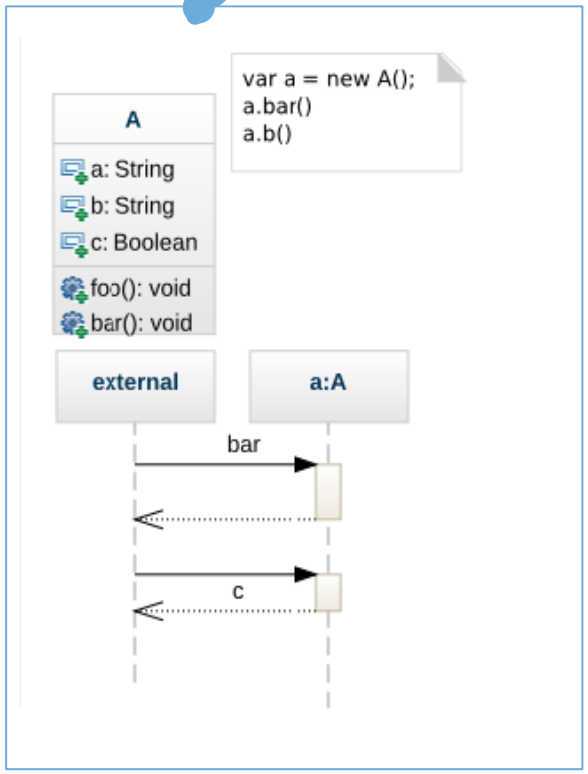
Built atop existent technologies => **using suboptimal tools when no option is available**

Aspects in Kermeta 3

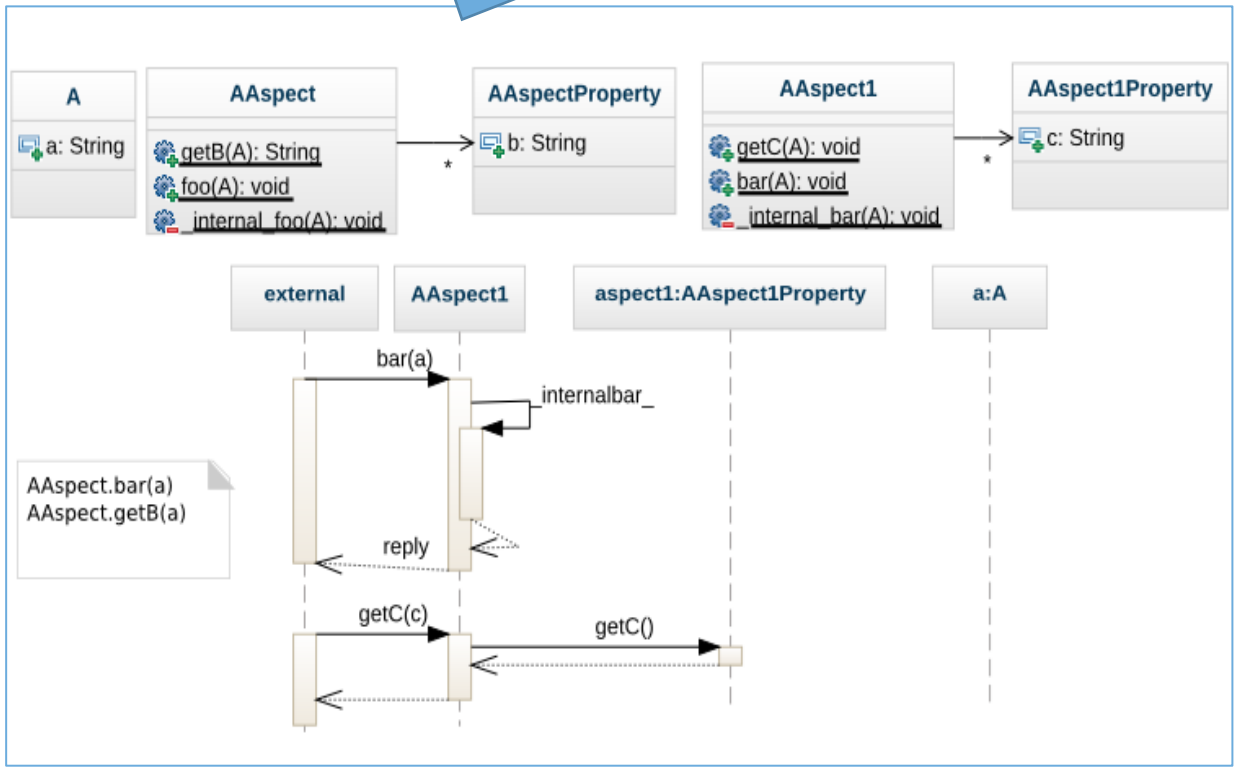
K3 definition



User's mental model



Real, "low-level", representation



State-of-the-art profilers' view?

Object	Class Name	Instance Size
A@32443234	**A**	**sizeof(A) + sizeof(A.a)**
…	…	…
AAspectProperty@fdsfdf3343	**AAspectProperty**	**sizeof(AAspectProperty) + sizeof(AAspectProperty.b)**
…	…	…
AAspect1Property@dfdsfkj566	**AAspect1Property**	**sizeof(AAspect1Property)**
…	…	….

Again … why custom memory profilers?

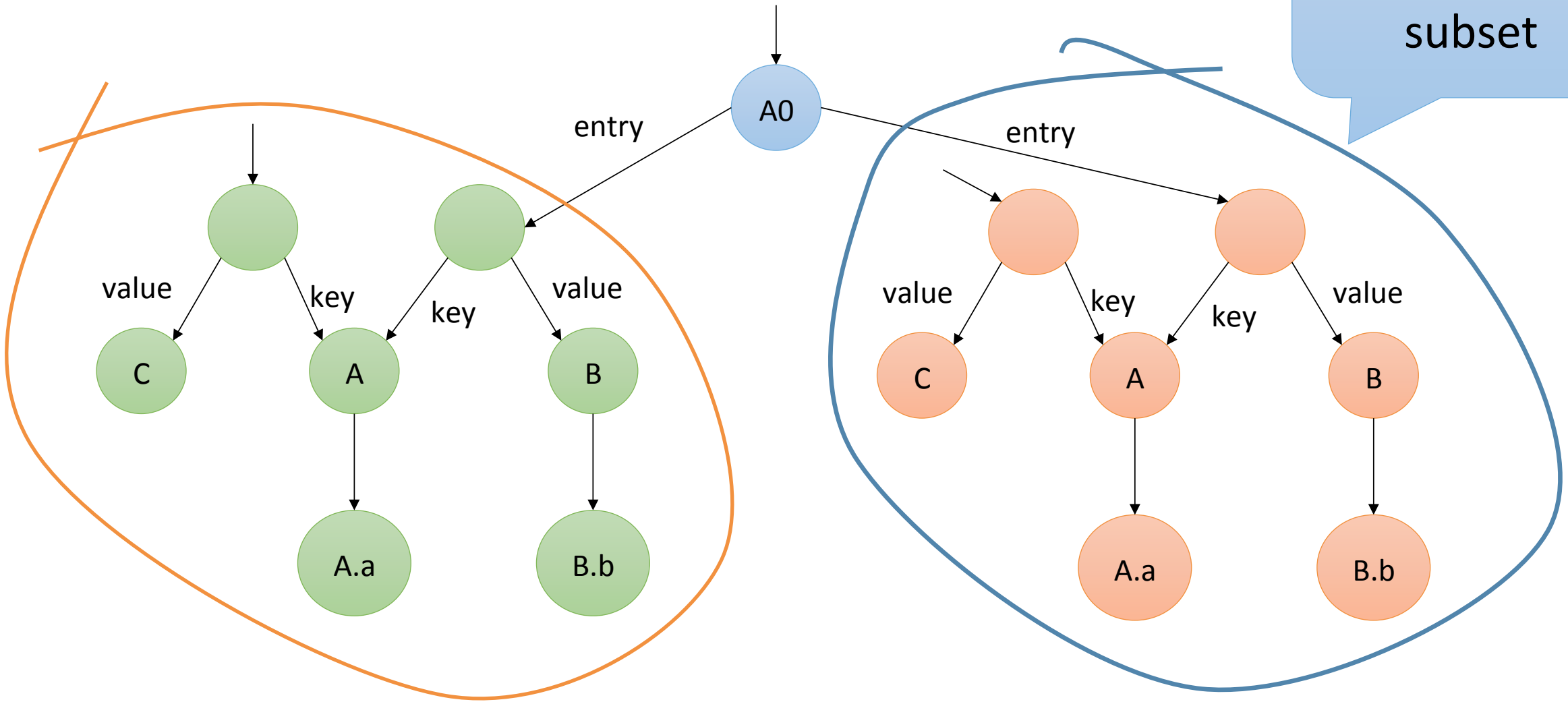
Memory profilers for Java mostly
offer support for standard
Java abstractions.

None at all or just restricted support for
additional abstractions

But when there is support, solutions perform poorly

This forbid their usage at runtime

What problem we face?



Compute values on each subset

Problem in detail

- The heap is a DAG with thousands of nodes.
- Such a DAG changes a lot over time
- A memory analysis is:
  1. Solve a subgraph matching problem (NP-complete)
  2. Compute values on the subgraph
- Most queries are tractable
  - OQL (Eclipse MAT, VisualVM)
  - Cypher (internal experiments)
- Tractable doesn't mean ready for production environments
  - Preprocessing is a bottleneck
  - Duplicating the DAG is a poor choice

Brief DSL's description

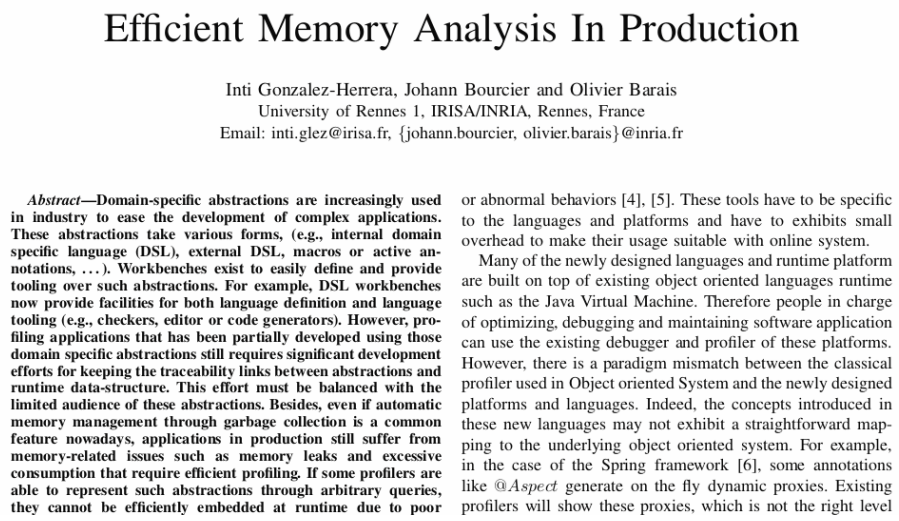
$$
\begin{aligned}
\mathit{Analysis} &= \{\mathit{Type}\}\ \{\mathit{Set}\}\ \{\mathit{Instance}\} \\
\mathit{Type} &= \mathit{id} : (\textbf{`table-of'}\ \mathit{id} \mid \textbf{`struct'}\ \{\mathit{Decl}\}\ \textbf{`end'}) \\
\mathit{Decl} &= \mathit{id} : (\mathit{id} \mid \textbf{`int'} \mid \textbf{`bool'} \mid \textbf{`string'}) \\
\mathit{Set} &= \textbf{`set-type'}\ s : \mathit{Property}\ \{\mathit{Property}\} \\
\mathit{Property} &= \mathit{DataID} \leftarrow e \\
\mathit{DataID} &= \textbf{`roots'} \mid \textbf{`membership'} \mid \\
&\quad \textbf{`on-inclusion'} \mid \mathit{Decl} \\
\mathit{Instance} &= \textbf{`instances-for'}\ \mathit{id}\ \textbf{`have-names'} = a \\
e &= r \mid b \wedge b \mid b \vee b \mid \neg b \\
r &= o \mid o\ \mathit{RelOp}\ o \mid o\ \textbf{`is'}\ \mathit{id} \mid \\
&\quad o \in (\textbf{`Unassigned'} \mid \textbf{`Entity'}) \\
o &= \mathit{Prefixed} \mid o + o \mid o - o \mid o * o \mid o/o \\
\mathit{Prefixed} &= -a \mid a\,[\text{`.'}\mathit{Member}] \\
\mathit{Member} &= \mathit{id}\,[\text{`('}e\text{`)'}]\,\{\text{`.'}\mathit{Member}\} \\
a &= \mathit{id} \mid \text{`('}e\text{`)'} \mid \mathit{Literal} \mid \\
&\quad \text{`['}\{\mathit{Decl}\}\text{`|'}\{\mathit{id} \leftarrow e\}\ [\textbf{`ret'}\ e]\text{`]'} \\
\mathit{Literal} &= \mathit{IntLit} \mid \mathit{StrLit} \mid \mathit{BoolLit} \mid \text{`\#['}\ \mathit{Lst\text{-}Expr}\ \text{`]'} \mid \\
&\quad \textbf{`struct'}\ \mathit{id}\ \mathit{Lst\text{-}Expr}\ \textbf{`end'} \\
\mathit{Lst\text{-}Expr} &= e\,\{\text{`,'}\ e\}
\end{aligned}
$$

Fig. 2. DSL syntax in a simplified EBNF notation; the operators for concatenation and termination are not used. Note that the non-terminals *id*, *IntLit*, *StrLit*, *BoolLit* are undefined. They correspond to identifiers, integer constants, string constants as in Java and the boolean constants.

- Subgraph specification
- User-defined data
- Collections
- Anonymous lambda expressions to operate on collections

DSL for custom memory profiling

- Trade-off between **expressiveness** and **performance**
- The computational model is simple and explicit
  - Linear time execution on the number of objects
  - Easy to see the cost of profiling
- Execute the DSL by traversing the DAG of live objects
- Support many queries, particularly resource consumption queries



Efficient Memory Analysis In Production

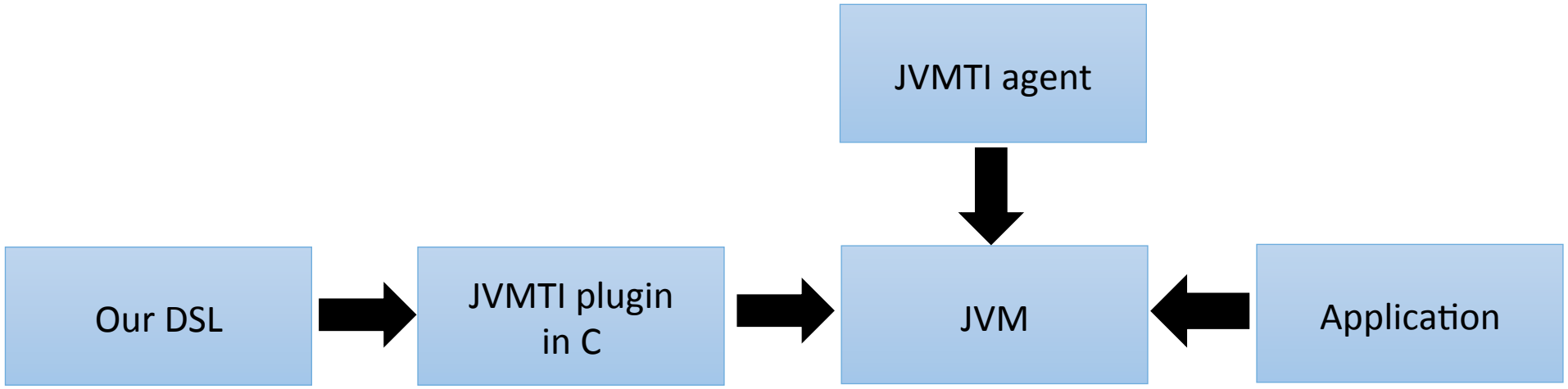
Inti Gonzalez-Herrera, Johann Bourcier and Olivier Barais
University of Rennes 1, IRISA/INRIA, Rennes, France
Email: inti.glez@irisa.fr, {johann.bourcier, olivier.barais}@inria.fr

***Abstract*—Domain-specific abstractions are increasingly used in industry to ease the development of complex applications. These abstractions take various forms, (e.g., internal domain specific language (DSL), external DSL, macros or active annotations, ...). Workbenches exist to easily define and provide tooling over such abstractions. For example, DSL workbenches now provide facilities for both language definition and language tooling (e.g., checkers, editor or code generators). However, profiling applications that has been partially developed using those domain specific abstractions still requires significant development efforts for keeping the traceability links between abstractions and runtime data-structure. This effort must be balanced with the limited audience of these abstractions. Besides, even if automatic memory management through garbage collection is a common feature nowadays, applications in production still suffer from memory-related issues such as memory leaks and excessive consumption that require efficient profiling. If some profilers are able to represent such abstractions through arbitrary queries, they cannot be efficiently embedded at runtime due to poor** or abnormal behaviors [4], [5]. These tools have to be specific to the languages and platforms and have to exhibits small overhead to make their usage suitable with online system.

Many of the newly designed languages and runtime platform are built on top of existing object oriented languages runtime such as the Java Virtual Machine. Therefore people in charge of optimizing, debugging and maintaining software application can use the existing debugger and profiler of these platforms. However, there is a paradigm mismatch between the classical profiler used in Object oriented System and the newly designed platforms and languages. Indeed, the concepts introduced in these new languages may not exhibit a straightforward mapping to the underlying object oriented system. For example, in the case of the Spring framework [6], some annotations like *@Aspect* generate on the fly dynamic proxies. Existing profilers will show these proxies, which is not the right level

Language implementation

- Java
- XText



- **Optimization is the main issue in the implementation**
  - Avoid precomputing unneeded data by using variability
  - Avoid traversing leaf nodes
  - Reordering expressions
  - Minimizing number of graph traversals

Evaluation

- Performance gain
  1. Impact of Analysis on the Total Execution Time
  2. Comparing Analysis Time for Simple Assertion
  3. Analysis Time in Real Scenarios
- Discussion on expressiveness

Impact of Analysis on the Total Execution Time

- Very simple analysis
- Execute the analysis many times
- Use different analysis techniques

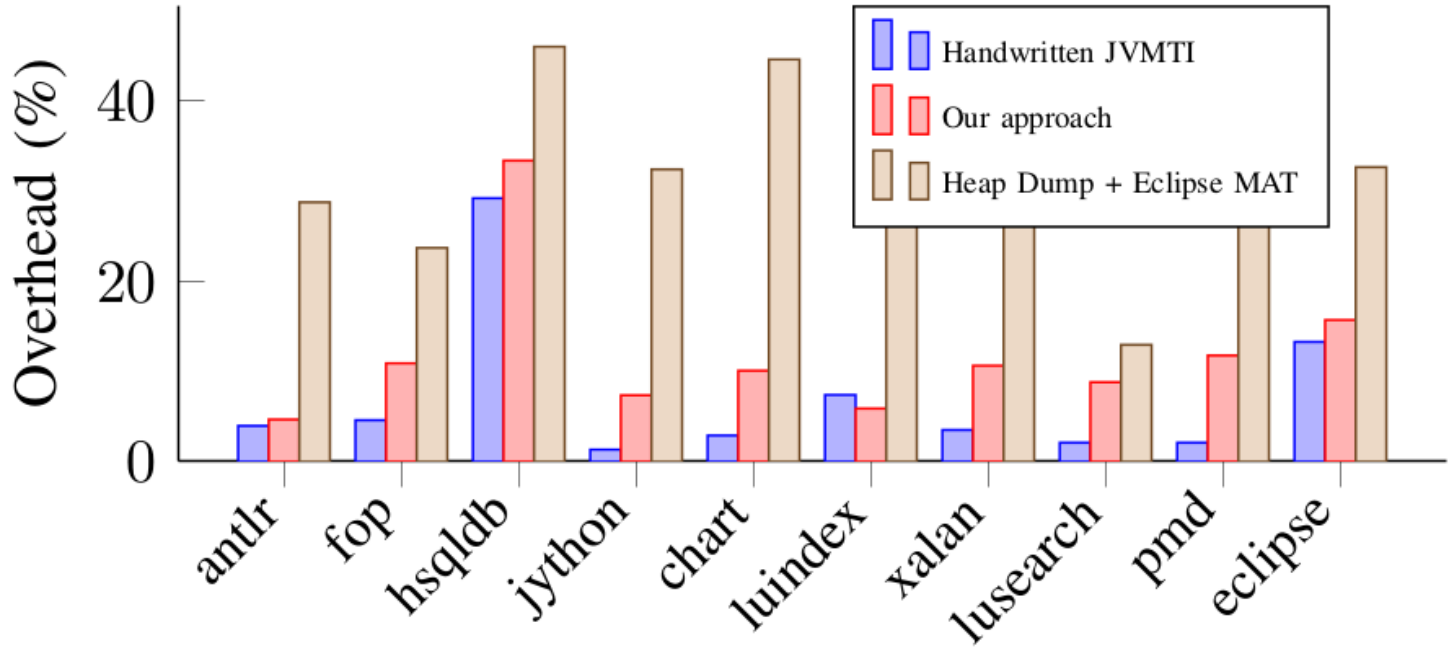


Fig. 3. Overhead on execution time compared to the execution without memory analysis for different tests in the DaCapo Benchmark

Comparing Analysis Time for Simple Assertion

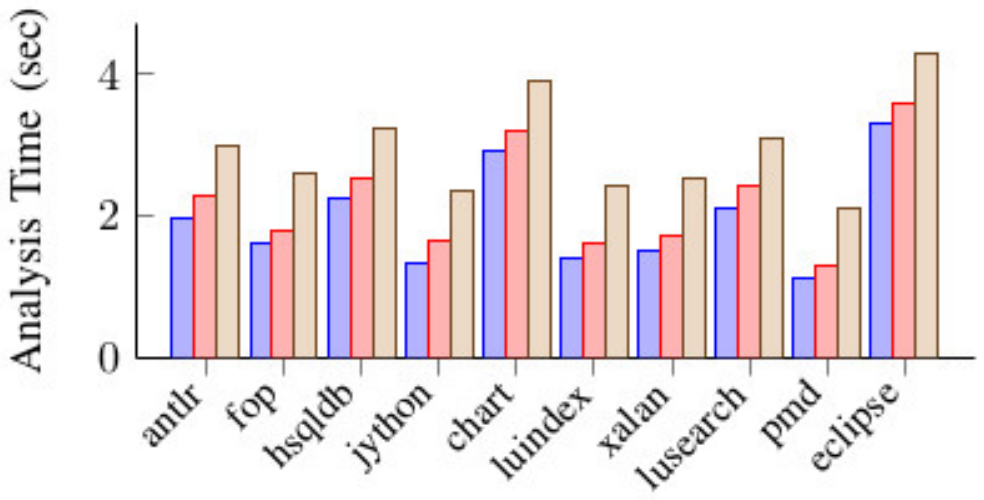


Fig. 4. Analysis time with default input size

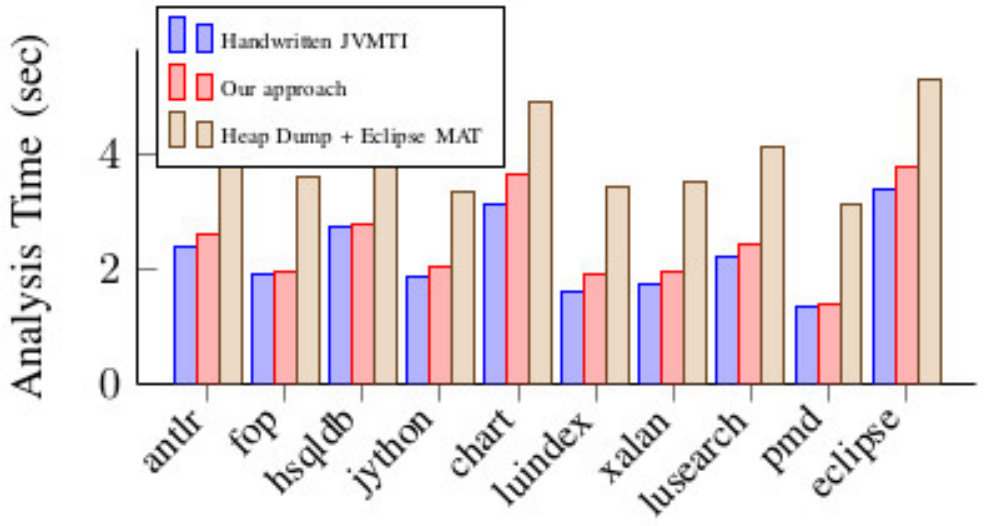


Fig. 5. Analysis time with large input size

Analysis Time in Real Scenarios

- OSGi-based applications
- Compute the memory consumption of each component

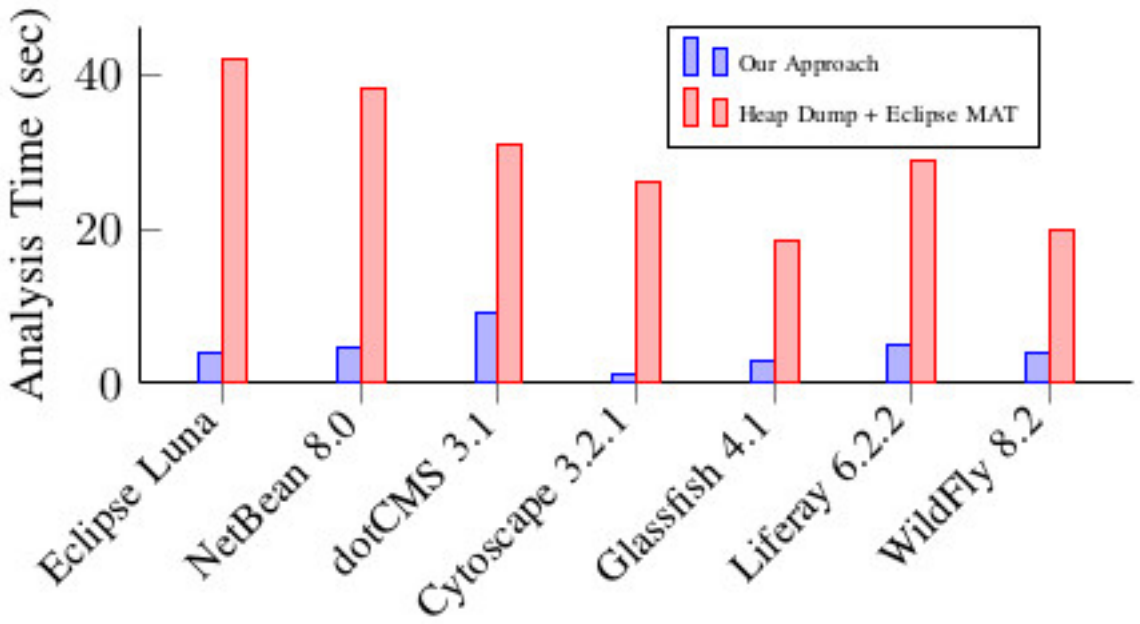
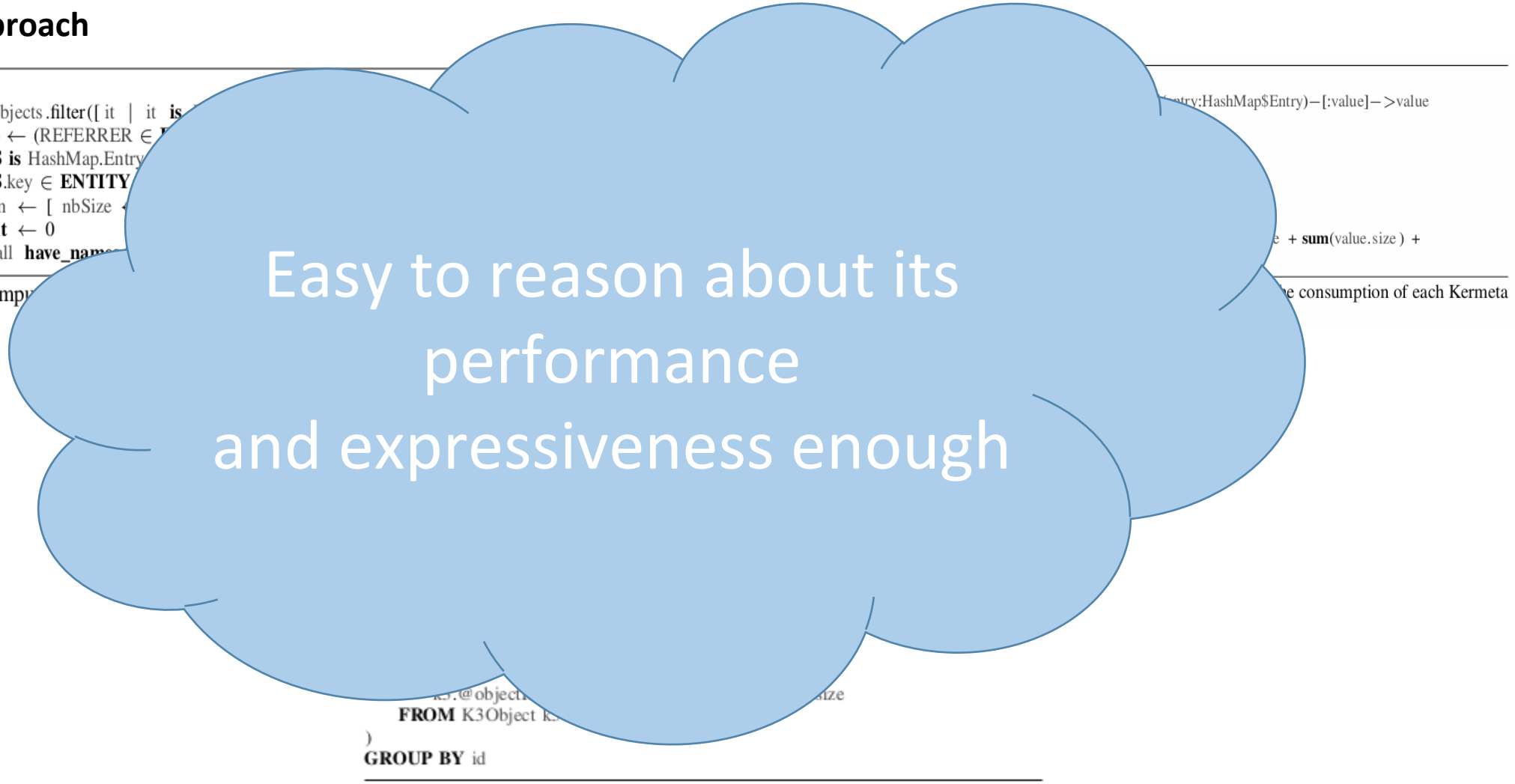


Fig. 6. Analysis time for real applications. It shows the time needed to compute an analysis just once. The analysis aims at finding the consumption of the top components as shown in listing 6.

Discussion on Language Expressiveness

Our approach



Conclusions and Future Works

- Tooling support for high-level abstractions is incomplete. We lack tools to deal with maintenance tasks.
- We provide a DSL to define custom memory profilers which show low overhead
  - Performance has been evaluated
  - Expressiveness was discussed
- For a new DSL, is it possible to automate the generation of a profiler?