

Incremental Execution of Name and Type Analysis

Introduction

Language workbenches are tools that support the efficient definition, reuse and composition of languages and integrated development environments (IDEs) [1]. We develop the Spoofox [2] Language Workbench, a workbench for developing textual languages with full IDE support in Eclipse.

IDEs provide a wide variety of language-specific editor services such as syntax highlighting, error marking, and code completion (see Figure 1) in real-time, while the program is edited. These services require syntactic and semantic analyses of the program. Thereby, timely availability of analysis results is essential for IDE responsiveness.

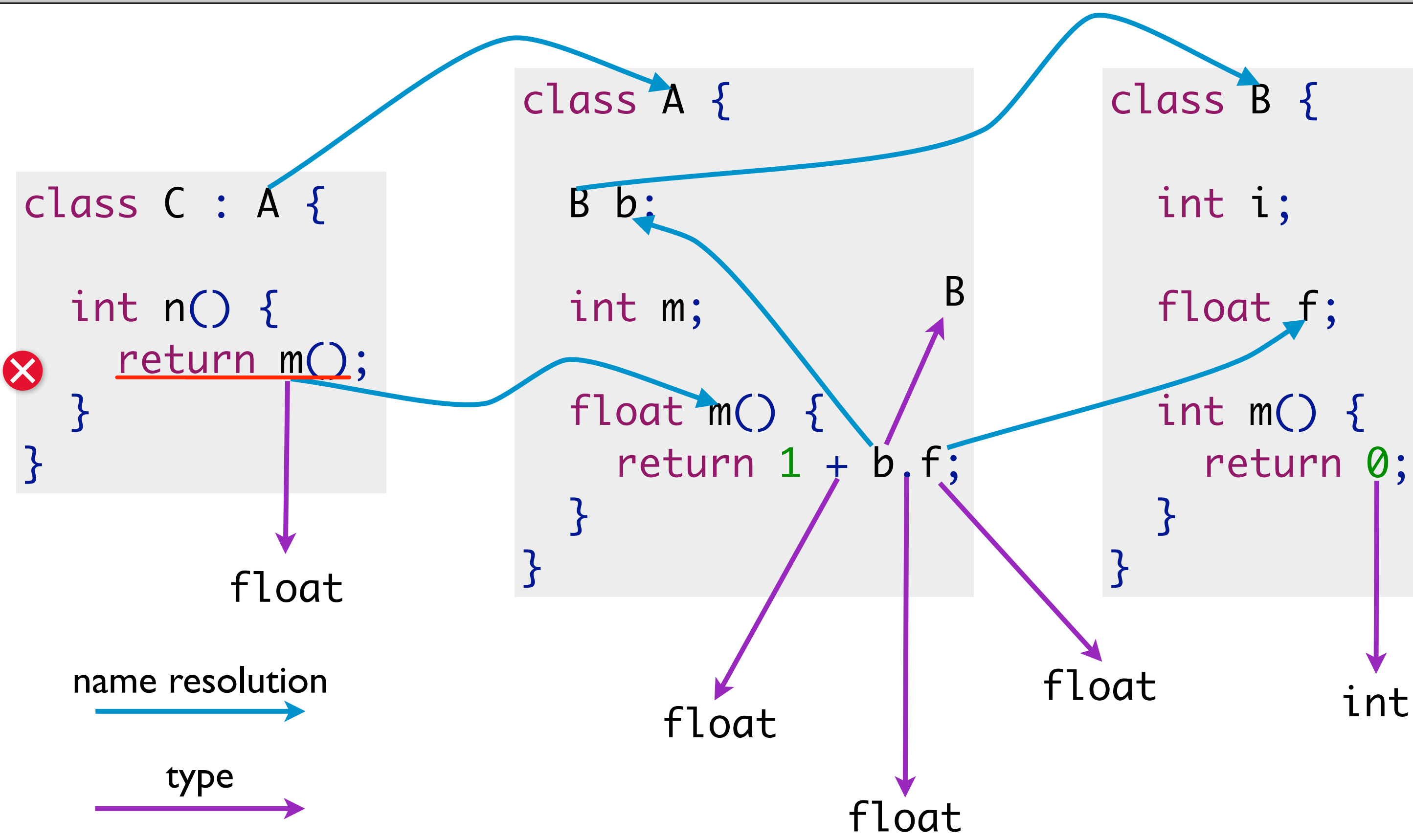
Whole-program analyses do not scale because the size of the program determines the performance of such analyses. Incremental analysis reuses previous analysis results of unchanged program parts and reanalyses only parts affected by changes. We focus on incremental name and type analysis, because it is required by many editor services.

```

1 class QuickSort {
2   public static void main(String[] a) {
3     System.out.println(new QS().Start(10));
4   }
5 }
6
7 class QS {
8   int[] number;
9   int size;
10  public int Start(int sz) {
11    int aux01;
12    aux01 = this.Init(sz);
13    System.out.println(9999);
14    aux01 = siz - 1;
15    aux01 = thi
16    return 0;
17  }
    
```

Figure 1. Source code editor in Spoofox with syntax highlighting and other editor services

Figure 2. Three C# files with name resolutions and type relations



Name and Type Analysis

The essence of name analysis is establishing relations between *definitions* that bind a name and *references* that uses that name. Type analysis is concerned with assigning a type to each expression in the program. Figure 2 on the left shows three C# files and their name and type relations.

There are many dependencies between and within these relations, even between files. For example, the type of the field access `b.f`; depends on the type of `b` and the type of `f`, which is defined in another file.

Whenever changes are made, relations need to be updated to reflect changes in the program. Complex dependency structures make incrementally updating these relations non-trivial.

Incremental Execution: Tasks

Instead of immediately executing name and type calculations when encountered in the program, we create deferred *analysis tasks* [3] that are executed at a later time. A task is a (small) unit of computation that can depend on other tasks, and can only be executed if all its dependencies have been executed.

From a program, a graph of name and type tasks, disconnected from the actual program, can be extracted. The task graph that is derived from the C# program can be seen in Figure 3 on the right.

The disconnection of tasks from the program means that we do not need to compare against the old program when changes occur. Instead, when a program changes, tasks are recollected and compared against the old set of tasks. Tasks that change have to be re-executed, as well as tasks that depend on changed tasks. Unchanged tasks are *not* re-executed, making name and type analysis incremental.

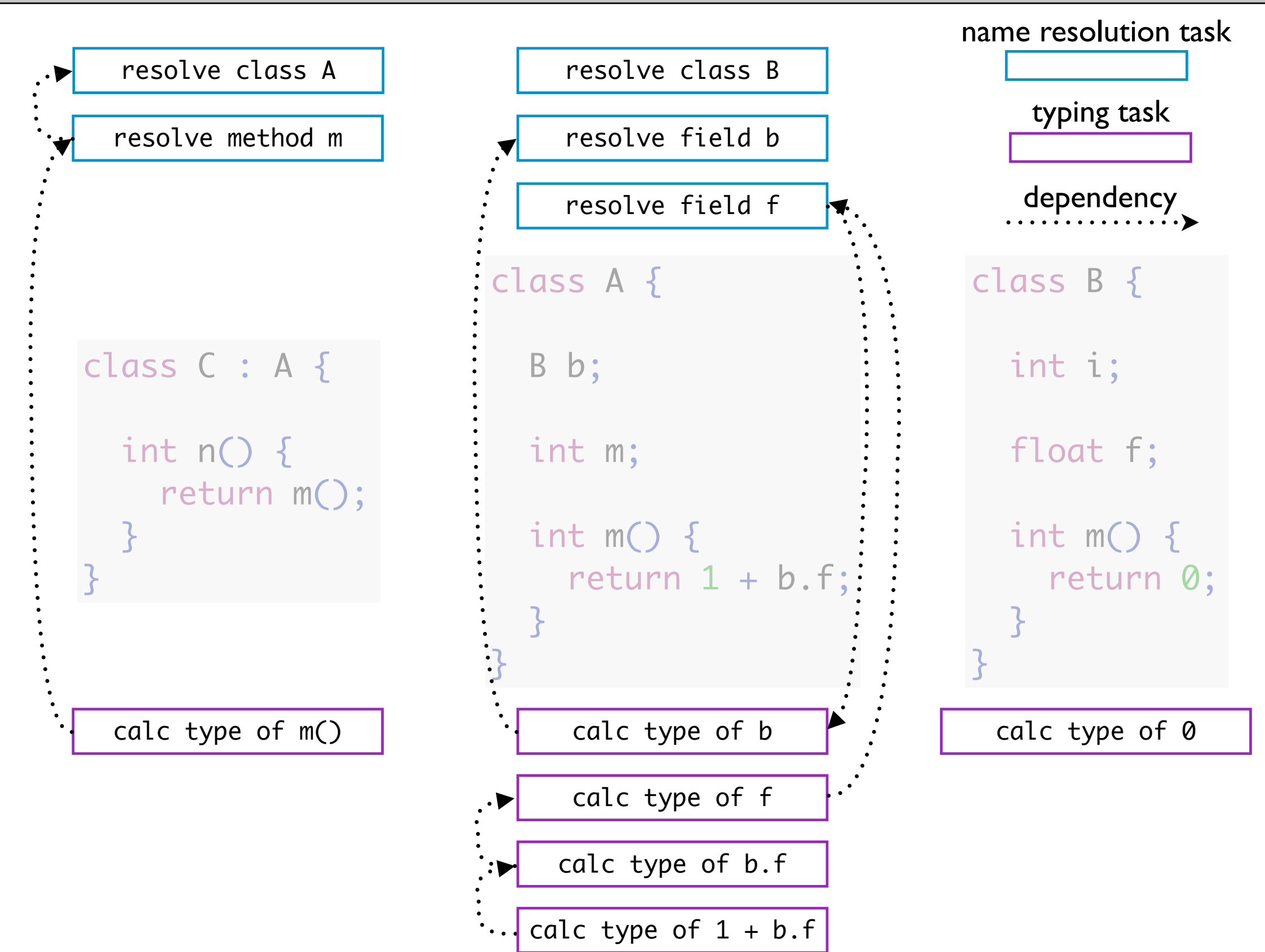


Figure 3. Name and type tasks with dependencies

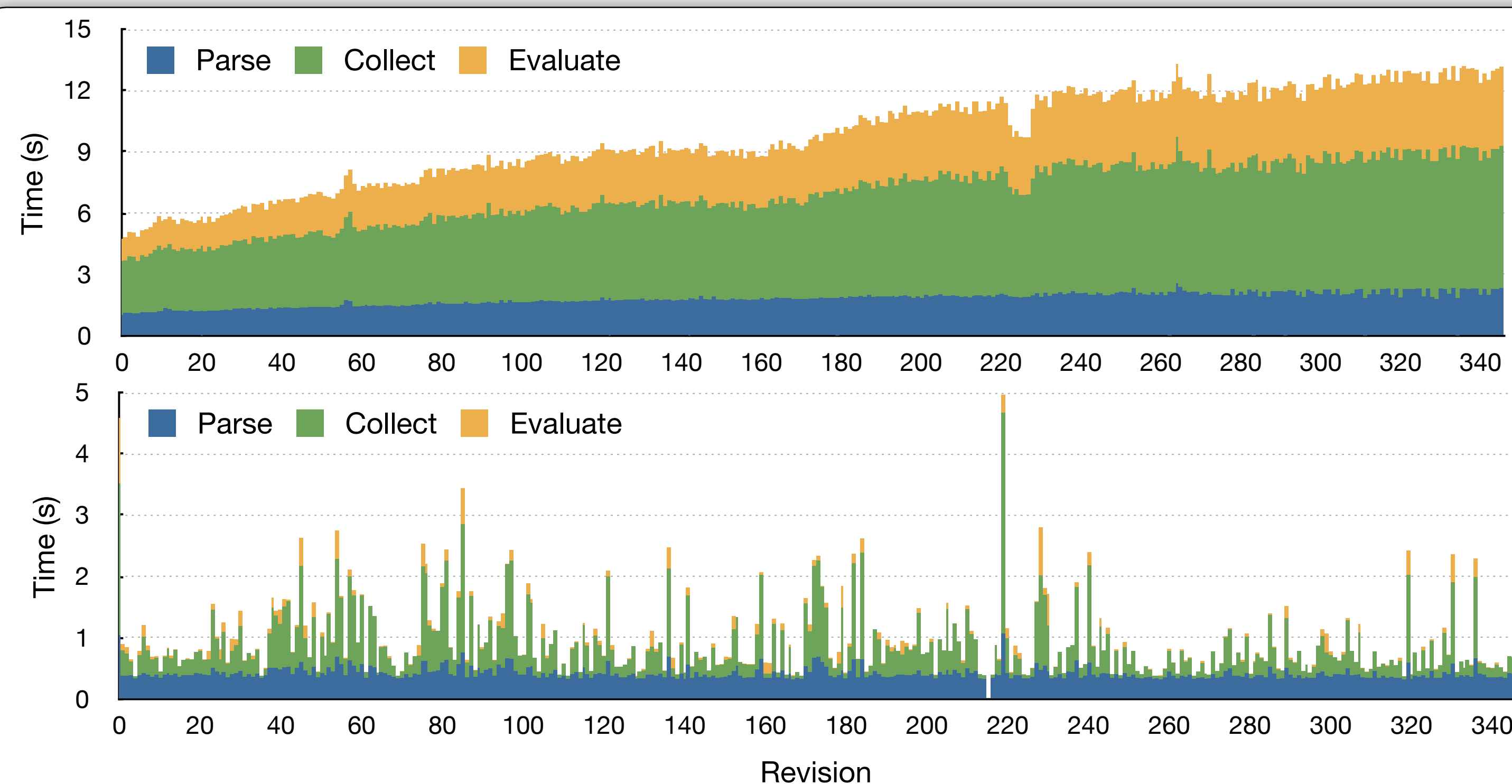


Figure 4. Non-incremental (top), and incremental (bottom) analysis time over project revisions

Performance Evaluation

To evaluate our approach we have re-implemented the name and type analysis of the WebDSL [4] language using tasks. WebDSL is a domain specific language for developing dynamic web applications.

We took the source code repository of Yellowgrass [5], an issue tracker written in WebDSL, and performed analysis for each revision in the repository. We measured performance for both non-incremental (full, from scratch) and incremental analysis, which can be seen in Figure 4. It is clear that full analysis scales with the project size, but incremental analysis does not. The correctness of incremental analysis was evaluated by comparing the results of the full analysis against the incremental analysis, which was equal for each revision.

The result is that incremental name and type analysis using tasks is fast enough for interactive usage in an IDE.

References

1. Sebastian Erdweg et al. The State of the Art in Language Workbenches. SLE (2013)
2. Lennart C. L. Kats et al. The Spoofox language workbench: rules for declarative specification of languages and IDEs. OOPSLA (2010)
3. Guido H. Wachsmuth et al. A Language Independent Task Engine for Incremental Name and Type Analysis. SLE (2013)
4. Danny M. Groenewegen et al. WebDSL: a domain-specific language for dynamic web applications. OOPSLA (2008)
5. Yellowgrass source code repository: <https://github.com/webdsl/yellowgrass>



Gabriël D. P. Konat
g.d.p.konat@tudelft.nl



Eelco Visser
e.visser@tudelft.nl



Guido H. Wachsmuth
g.h.wachsmuth@tudelft.nl

Vlad A. Vergu
v.a.vergu@tudelft.nl

Danny M. Groenewegen
d.m.groenewegen@tudelft.nl