Declarative Specification and 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 *Spoofax* [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.

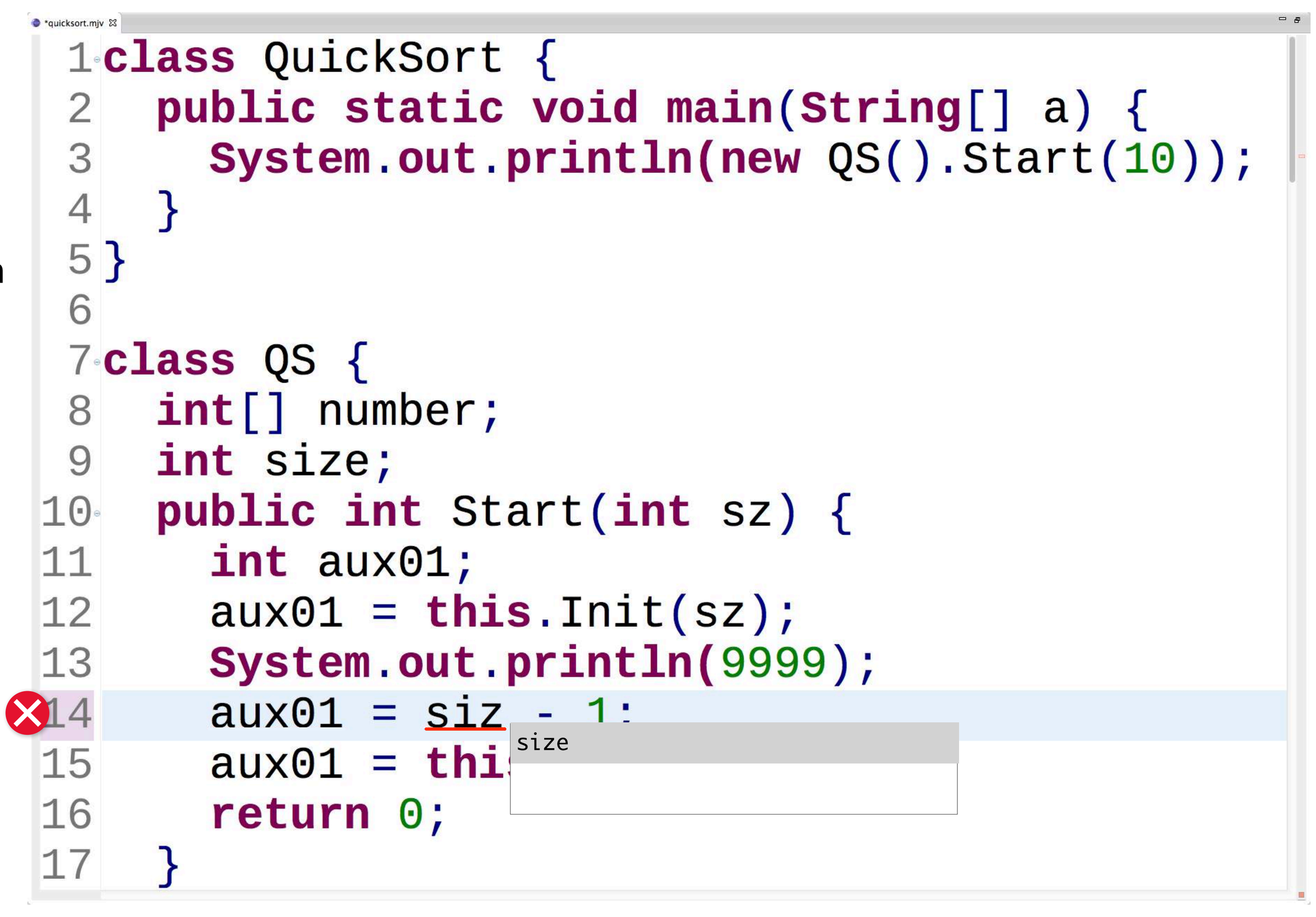


Figure 1. Source code editor in Spoofax with syntax highlighting, error marking, and code completion editor services

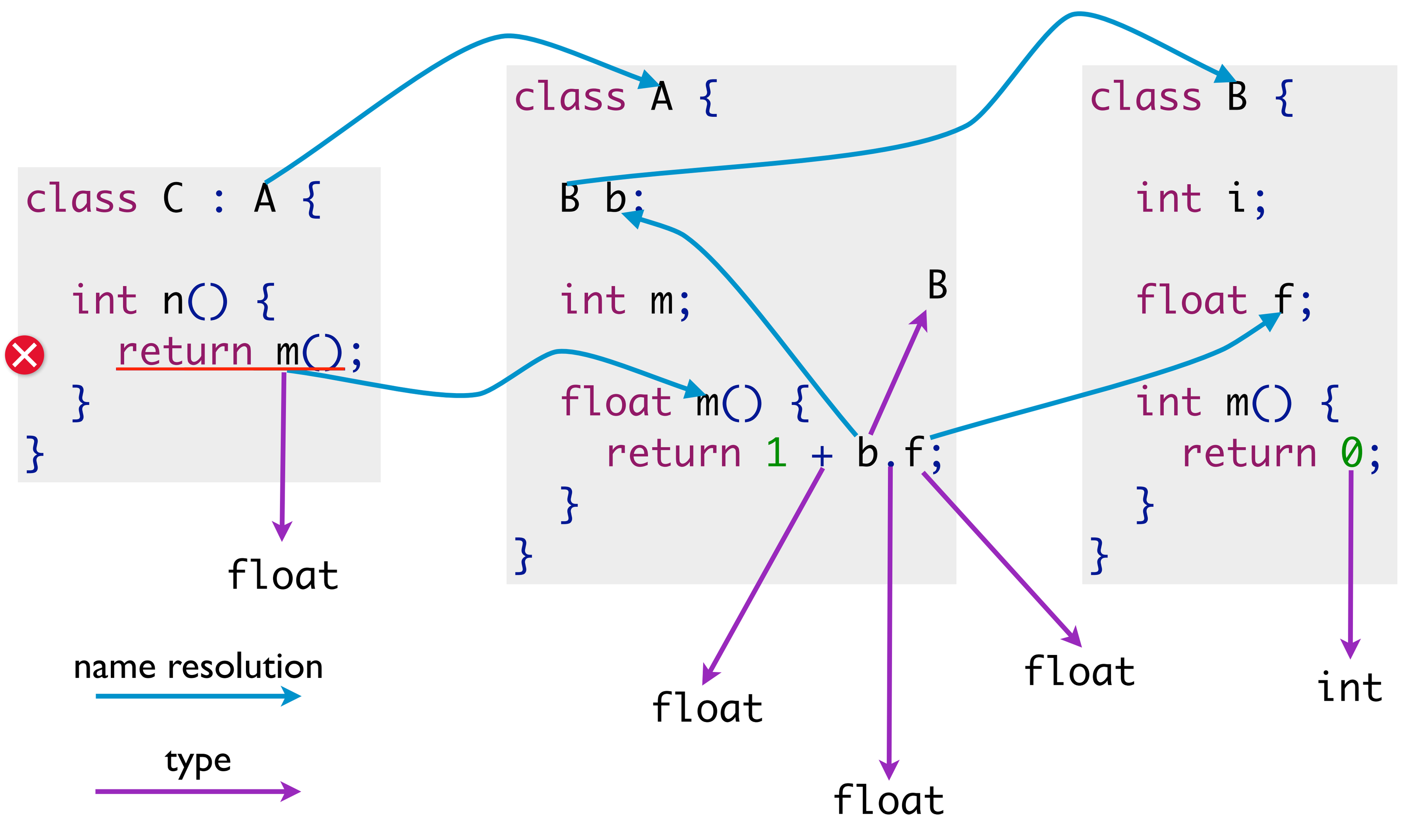


Figure 2. Three C# files with name resolution and type relations

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 unit of computation that can depend on other tasks, and can only be executed if all dependencies have been executed.

From a program, a graph of name and type tasks can be extracted which is not connected to the program any more. The task graph that is derived from the C# programs can be seen in Figure 3 on the right.

This disconnection of tasks from the program means that we do not need to compare against the old program when changes occur. Instead, when a file 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.

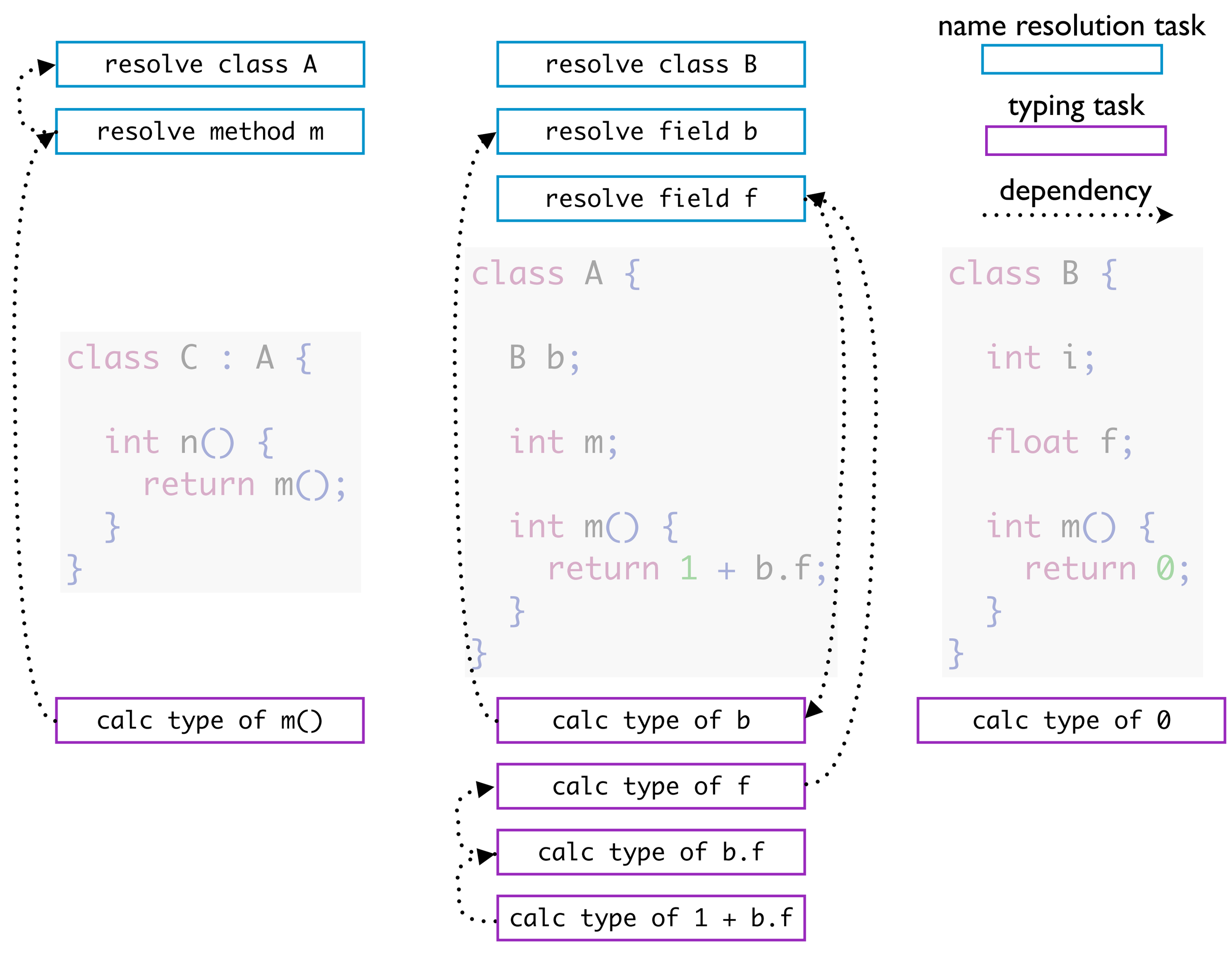


Figure 3. Name and type tasks with dependencies

```
binding rules

Class(c, _) :
  defines Class c
  scopes Field, Function

FieldDef(t, f) :
  defines Field f of type t

FieldAccess(exp, f) :
  refers to Field f in Class t
    where exp has type t

VarDef(t, v, _) :
  defines Variable v of type t
    in subsequent scope

VarRef(r) :
          refers to Variable r
  otherwise refers to Field r
```

Figure 4. Fragment of C# name binding specification in the Name Binding Language

```
type rules

IntLit(_) : IntType()
True()    : BoolType()
False()   : BoolType()

Add(x, y) : IntType()
  where x : x-ty
  and   y : y-ty
  and   x-ty == IntType()
    else error "expected integer" on x-ty
  and   y-ty == IntType()
    else error "expected integer" on x-ty

VarRef(v) : t
  where definition of v : t

Assign(exp, val) : exp-ty
  where exp : exp-ty
  and   val : val-ty
  and   exp-ty == val-ty
    else error "type mismatch" on val-ty
```

Figure 5. Fragment of C# type system specification in the Type System language

Declarative Specification

Tasks take care of the incremental execution of name and type analysis, this shifts the problem to collecting those tasks. We collect tasks automatically by *specifying* name and type analysis *declaratively.*

The Name Binding Language (NaBL) [4] is a metalanguage for declaratively specifying name binding and scope rules in terms of definitions, references, namespaces, scopes and imports. The Type System language (TS) is a metalanguage for declaratively specifying the type system.

Both languages use rules that pattern match the abstract syntax tree and assign name and type concepts to the program by constructing tasks. See Figures 4 and 5 for example programs in NaBL and TS, respectively. Using these specifications, tasks are automatically collected.

Results

To evaluate our approach we have re-implemented the name and type analysis of the WebDSL [5] language using the outlined approach. WebDSL is a domain specific language for developing dynamic web applications.

We took the source code repository of Yellowgrass [6], an issue tracker written in WebDSL, and performed analysis for each revision in the repository. We measured performance for both full and incremental analysis, which can be seen in Figures 6 and 7. 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.

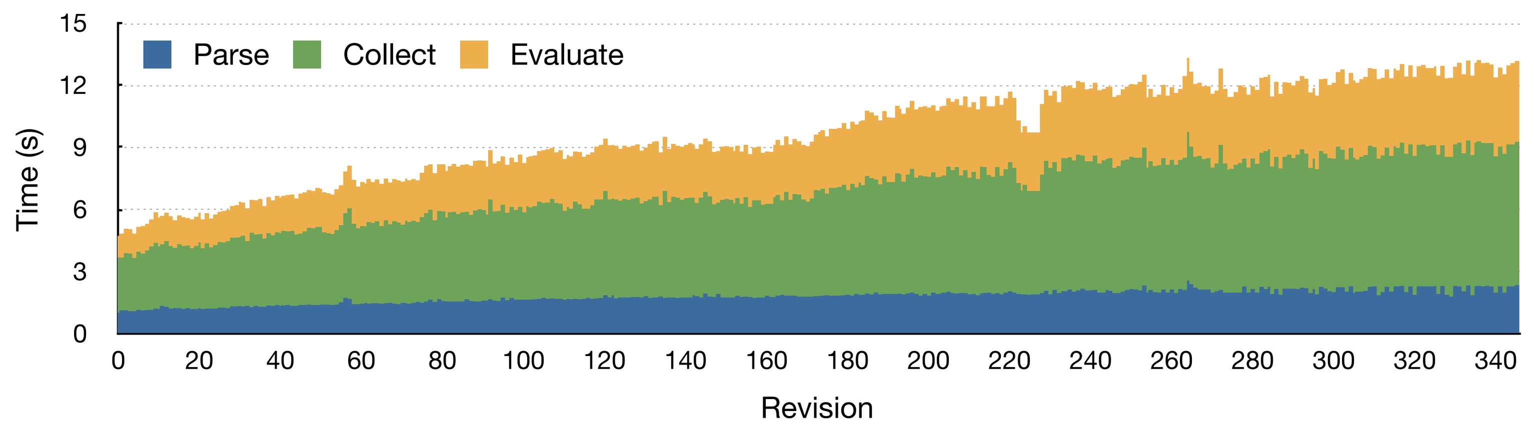


Figure 6. Full, non-incremental analysis time over project revisions

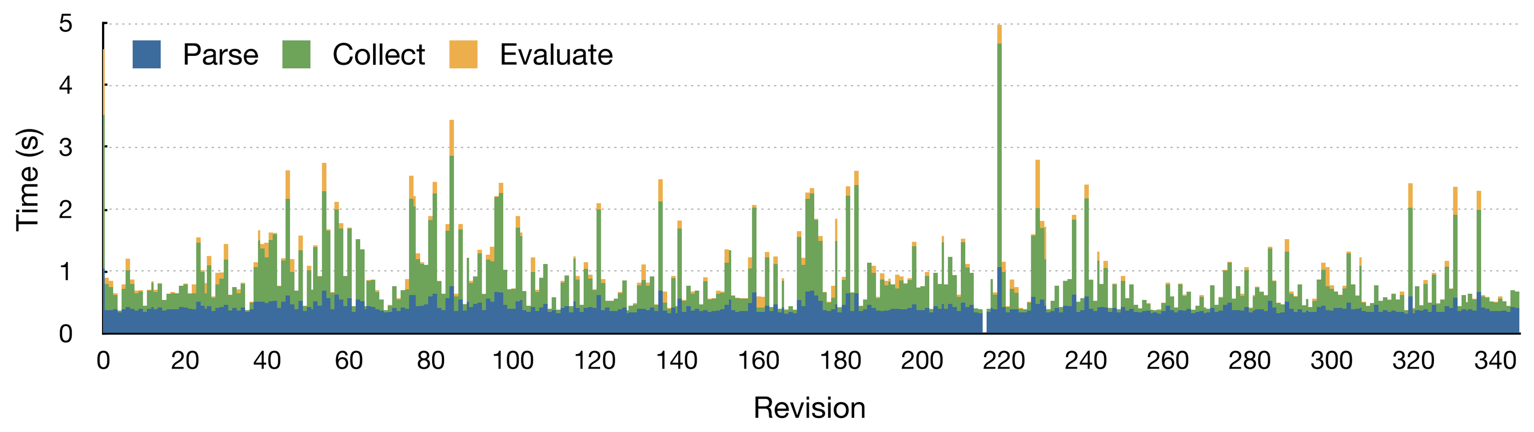
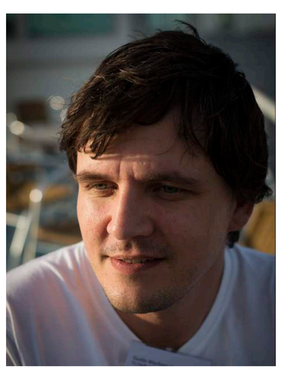


Figure 7. Incremental analysis time over project revisions

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

References

1. Sebastian Erdweg et al. The State of the Art in Language Workbenches. SLE (2013)
2. Lennart C. L. Kats et al. The Spoofax 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. Gabriël D. P. Konat et al. Language-parametric name resolution based on declarative name binding and scope rules. SLE (2012)
5. Danny M. Groenewegen et al. WebDSL: a domain-specific language for dynamic web applications. OOPSLA (2008)
6. Yellowgrass source code repository: https://github.com/webdsl/yellowgrass

http://www.SPOOFAX.org

TU Delft
Delft University of Technology