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 analyses reuse previous analysis results where possible 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.png)

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

Name and Type Analysis

The essence of name analysis is establishing relations between definitions that bind a name and references that uses that name and find a location where 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 t.f depends on the type of t 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.

![Figure 2](csharp_files.png)

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

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 with the old program when changes occur. Instead, when the tasks are re-executed 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.

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 (WebDL) [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.

![Figure 3](csharp_tasks.png)

**Figure 3.** C# home and type tasks with dependencies.

![Figure 4](naBL.png)

**Figure 4.** Fragment of NaBL name binding specification in the NaBL Language.

![Figure 5](TS.png)

**Figure 5.** Fragment of TS type system specification in the Type System language.

Results

To evaluate our approach we have reimplemented 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 analyzed performance 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](webDSL.png)

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

![Figure 7](webDSL_incremental.png)

**Figure 7.** Incremental analysis time over project revisions.

References

6. Yellowgrass source code repository: https://github.com/eggto/hellowgrass
7. TUDelft University Technology.