From Whole Program Compilation to Incremental Compilation: A Critical Case

Jeff Smits* j.smits-1@tudelft.nl

Gabriël Konat* g.d.p.konat@tudelft.nl

Eelco Visser* e.visser@tudelft.nl

*Delft University of Technology

Introduction Compilation time of a software project is an important factor in how easily the project can be changed. When the compilation time is low, it is cheap to experiments with changes to the project, which can be tested immediately after recompilation. Therefore, fast recompilation has been a topic of interest for a long time [1].

One way to speed up recompilation is to save intermediate results during compilation. If parts of the program do not change, then the intermediate results of those parts can be reused. The term *separate compilation* applies to compilation where intermediate results are saved per file [4]. For sub-file level tracking of changes and intermediate results, the term *incremental compilation* is used [8, 7].

Although incremental compilation has clear benefits for recompilation speed, it is not trivial to implement one. Language features can sometimes make incremental compilation difficult, because they may allow one piece of code to influence compilation of a distant piece of code. Such cross-file definitions have forced some languages to use a whole-program compiler, which scales linearly or worse with the size of a project.

We introduce a design approach for incremental compilers that we believe may be applicable to other languages. We demonstrate it on the critical case of Stratego [3, 2], a term rewriting language with open extensibility features. After a brief overview of the open extensibility features, we show our compilation method, which is somewhere in between separate and incremental compilation. Our approach allows us to reuse almost all of the existing compiler while gaining great improvements in recompilation speed. We evaluate the new compiler with a benchmark on the version control history of a large Stratego project.

Stratego Stratego is a tree transformation language with rewrite rules and strategies. Such rules and strategies are defined by name, and multiple definitions with the same name are merged as different options of the rule or strategy. This works over different files, which is used as an extensibility mechanism.

This extensibility mechanism can, for example, be used to desugar language features to a core language. The core language can be defined in one module with an identity desugaring, and different extensions of the language and the corresponding desugaring can be written in their own module. To support this extensibility mechanism, the Stratego compiler has always used whole-program compilation.

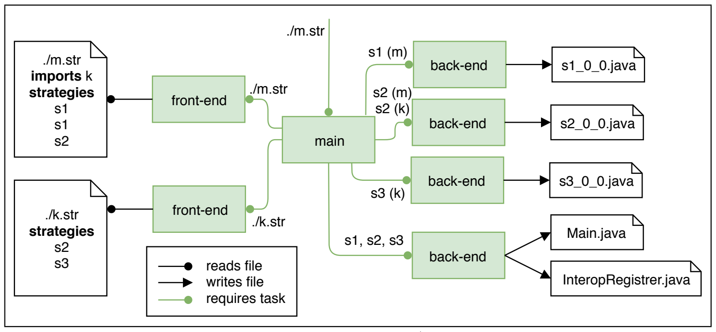

Figure 1: Static linking model. A frontend task for each file, backend task for each strategy.

Incremental Compilation To achieve incremental compilation, we split up the compiler into a front-end and a back-end[1]. The front-end is adapted so it can be called separately on each file. It no longer does static checks that require information from other files. The front-end then generates an intermediate representation for each strategy, and the necessary information that can be used to do the static checks that were removed. The static checking information is aggregated for all files. Checking is done once all files have been processed by a front-end task.

The back-end is adapted so it can be called separately on each strategy. We first merge definitions of the same name from different front-ends. Then we can apply the adapted back-end code to produce the expected Java class for each definition.

See Figure 1 for a diagram of the front- and back-end tasks on an example. The `main` task orchestrates everything and receives the main file as input. It starts a front-end task for the main file, and receives the processed information of the file. It uses imports to find more files to process with front-end tasks. The static checks are done within the main task. After the static checks, back-end tasks are run for each

[1]This was already the internal architecture of the compiler.

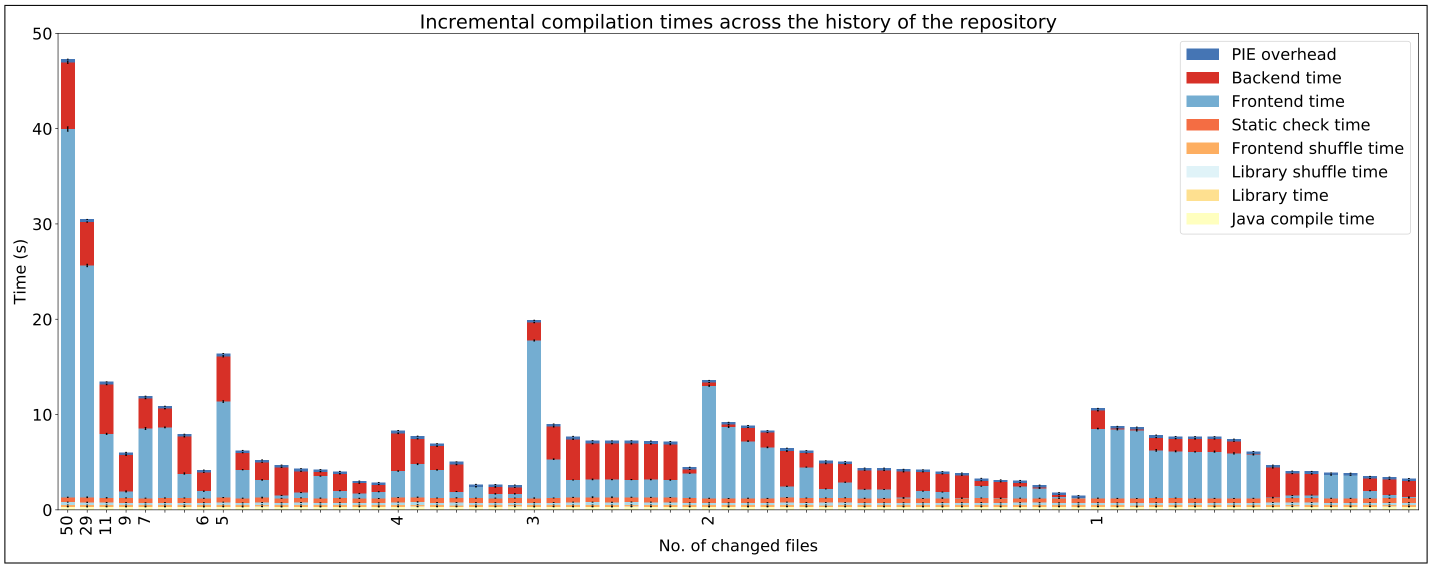

Figure 2: Benchmark results for WebDSL commit history. The tail end of 1 file commits has been truncated to make the figure more readable. The truncated tail has 50 more 1 file commits that taper off to 2 seconds total time.

strategy. A last back-end task generates some boilerplate classes based on a list of all strategies.

The network of tasks is created with the PIE incremental build system [6, 5]. PIE incrementally executes affected tasks when given a set of changed files. We characterise our new compiler as an incremental compiler, but perhaps a more accurate description is a multi-stage separate compiler. It applies separate compilation in that it will process a whole file to intermediate representation when part of the file is changed, through a front-end task. However, it only generates code for actually changed definitions, through a back-end task. This resembles incremental compilation and has much of the benefits.

Design Approach The general design approach here is to split up files into units that can be processed independently. The earlier this split is made, the less other stages need to redo when a one of those units is changed in a file. In the case of Stratego, top-level strategy definitions are the independent units.

These independent units can be recombined later when necessary. In Stratego's case this is before code generation, as each strategy of the same name has to be combined into one Java class in the compiler output. Notably, the units need not have come from the same original file, which is exactly what we need to compile Stratego.

Of course the units can still have some influence on each other, most likely during static analysis. For Stratego we simply added all information for static analysis together. This means that the static analysis is not incremental. Early measurements already showed that the static analysis is not a large factor in the total compile time, therefore we did not pursue a more incremental solution. Such a solution would be difficult to express in PIE anyway, as Stratego's imports may form cycles, and easily do so because they are transitive. As task dependencies may not have cycles in PIE, we would have to work around that limitation in our task model. But other languages may lend themselves better to incremental static analysis. For example, Modula-2 produces static analysis summaries for each input file [9].

Evaluation To evaluate our new compiler for Stratego, we benchmark a large Stratego project (27 KLOC excluding whitespace and comments). We replay its commits from version control and run our incremental compiler for each commit.

We run this benchmark in a virtual machine so the machine image can be easily reused by others to rerun our experiments. The virtualisation makes the absolute time measurements slightly higher than when run natively. The virtual machine runs on a MacBook Pro (Early 2013), with an Intel Core i7 2.8GHz CPU, 16 GB 1600 MHz DDR3 memory and an SSD harddisk.

Figure 2 shows the results of the incremental runs on commits of the history. We order the commits by number of changed files, seen on the x-axis. The y-axis values show how all compilations took under 50 seconds, and the more likely scenarios for an incremental compiler—up to 10 changed files—takes between 2 and 20 seconds. The original Stratego compiler takes roughly 130 seconds to compile the entire project, and, crucially, takes that amount of time for every commit.

Conclusion We have introduced a design approach for incremental compilers that fits languages with cross-file definitions. We demonstrated this approach on the critical case of Stratego. The result is a multi-stage separate/incremental compiler that has a 2.6-65x speedup for compilation over the last 2 years of history of a large Stratego project.

References

[1] John Warner Backus and William P. Heising. “Fortran”. In: *TC* 13.4 (1964), pp. 382–385. DOI: http://dx.doi.org/10.1109/PGEC.1964.263818.

[2] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. “Program Transformation with Scoped Dynamic Rewrite Rules”. In: *Fundamenta Informaticae* 69.1-2 (2006), pp. 123–178. DOI: https://content.iospress.com/articles/fundamenta-informaticae/fi69-1-2-06.

[3] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. “Stratego/XT 0.17. A language and toolset for program transformation”. In: *Science of Computer Programming* 72.1-2 (2008), pp. 52–70. DOI: http://dx.doi.org/10.1016/j.scico.2007.11.003.

[4] Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite. “Early Experience with Mesa”. In: *CACM* 20.8 (1977), pp. 540–553.

[5] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. “Scalable incremental building with dynamic task dependencies”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, 2018, pp. 76–86. DOI: https://doi.org/10.1145/3238147.3238196.

[6] Gabriël Konat, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. “PIE: A Domain-Specific Language for Interactive Software Development Pipelines”. In: *Programming Journal* 2.3 (2018), p. 9. DOI: https://doi.org/10.22152/programming-journal.org/2018/2/9.

[7] Steven P. Reiss. “An approach to incremental compilation”. In: *sigplan*. 1984, pp. 144–156. DOI: http://doi.acm.org/10.1145/502874.502889.

[8] James L. Ryan, Richard L. Crandall, and Marion C. Medwedeff. “A conversational system for incremental compilation and execution in a time-sharing environment”. In: *afips*. 1966, pp. 1–21. DOI: http://doi.acm.org/10.1145/1464291.1464293.

[9] Niklaus Wirth. “Modula-2 and Oberon”. In: *HOPL*. 2007, pp. 1–10. DOI: http://doi.acm.org/10.1145/1238844.1238847.