Towards Incremental Compilation for Stratego

Jeff Smits
TU Delft
j.smits-1@tudelft.nl

Eelco Visser
TU Delft
e.visser@tudelft.nl

Abstract

Stratego is a transformation language based on term rewriting with programmable rewriting strategies. A program in Stratego consists of named rewrite rules and strategies. When definitions have the same name, they contribute to the same rule. This works across files, thereby allowing extensibility.

Due to this distribution of rules over modules, the Stratego compiler has always been a whole program compiler. Large Stratego programs are slow to compile as a result. In this work we present our approach to incremental compilation of Stratego. The approach may be useful for incremental compilation of other languages with similar cross-file features.

Keywords separate compilation, linking, Stratego

ACM Reference Format:
Jeff Smits and Eelco Visser. 2018. Towards Incremental Compilation for Stratego. In *Proceedings of 2018 ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity - Poster session (SPLASH Posters'18).* ACM, New York, NY, USA, 2 pages.

1 Introduction

Stratego [2, 3] is a tree transformation language with rewrite rules and strategies. It defines such rules and strategies by name. Multiple definitions with the same name are merged as different options of the rule or strategy. This works over different files, which is used 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 compiler compiles the whole program at once. For larger programs this is a slow process. Therefore we want to introduce incremental compilation for Stratego. This does, however, require some work as a number of cross-module features of Stratego need to be taken into account.

We present three different compilation models, the whole program one, and two incremental ones. We have found solutions to issues that are likely not unique to Stratego and may be reused for incremental compilation of other languages.

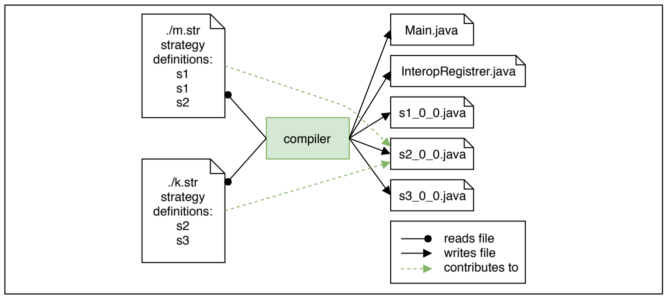

Figure 1. Whole-program compilation model. All modules are merged by a single process.

2 Compilation Models

The three different compilation models we present are: the slow whole-program compilation model, a dynamic linking compilation model, and a static linking compilation model.

Whole Program Whole program compilation takes in all relevant files with Stratego modules. It parses all and builds a single internal model of the program. This is then used to generate a Java class for each strategy, and to generate two shared classes. We illustrate this in Figure 1 for two modules. The `InteropRegistrer` class has a list of all strategy names to register to their implementations at run-time. The `Main` class is an entry-point for the program if the Stratego code is used stand-alone, and has some constants and other objects pre-allocated in static fields, which are then used in the other Java classes for the strategies. Strategies `s1` and `s3` are only defined in one module and each have their corresponding Java class in the compiler output. Strategy `s2` is defined in both modules, and merged by the compiler into a single Java class.

Dynamic Linking With a dynamic linking approach we run the compiler once for each module. This means that if multiple modules define the same strategy, we now have duplicate classes. These classes need to be merged (linked) at run-time. The Stratego runtime and the generated code need to be adapted so this dynamic linking can happen. This is illustrated in Figure 2. This model was first used in a case study of the Stratego compiler for the Pluto incremental build system [1].

The downsides of this approach are twofold: First, linking at run-time has an overhead during execution. Second, the Stratego runtime and generated code need to be adapted to support dynamic linking. This means they become incompatible with compiled Stratego code from earlier versions. That

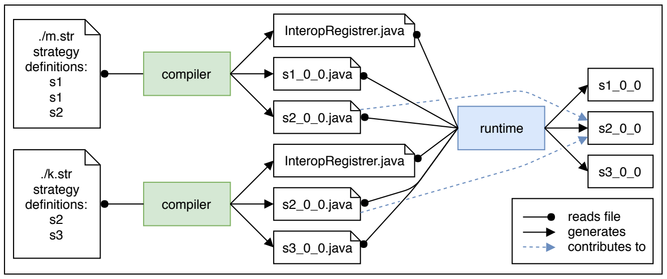

Figure 2. Dynamic linking model. Compiler run for each module, runtime merges strategies.

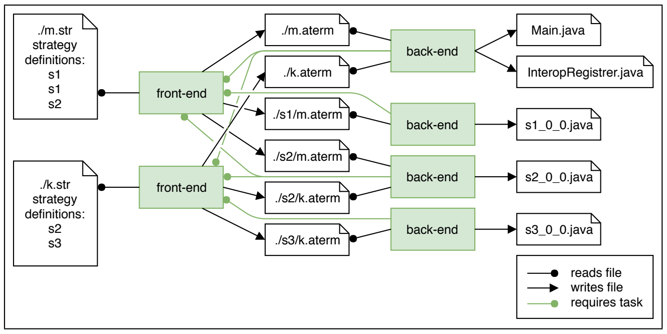

Figure 3. Static linking model. Frontend task for each module, backend task for each strategy.

incompatibility was the main reason we could not adopt this approach: We have some compiled Stratego code without the source code, therefore the runtime needs to be backward compatible.

Static Linking To achieve incremental compilation with static linking, we split up the compiler into a front-end and a back-end[1]. See Figure 3 for another illustration. The front-end can be called separately on each module. It generates a file for each strategy, named so that multiple front-end tasks generate files for the same strategy in the same directory. Now we have a directory of files for each strategy, which is taken as input to the back-end. The back-end merges all contributions to a strategy and generates one java class.

Note that we also expected a `Main` and `InteropRegistrer` class. These are generated separately based on extra information extracted from each module. For simplicity, we removed the caching of objects in static fields of `Main`, but the `InteropRegistrer` is unchanged. The list of strategies used in the `InteropRegistrer` is provided through the extra information file.

In this model, a change to a single strategy definition in a single module will result in some checks by the incremental system and the execution of one front-end task for the module and one back-end task for the changed strategy.

Reusable pattern We believe that there is a pattern, where one extracts extra information per module and merges it with another task. For example, we may use this pattern to extract overlays, a feature in Stratego for aliases of terms. These can be extracted for all modules, combined, and then given as an input to all other back-end tasks.

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

3 Future Work

In this section we briefly discuss work we still mean to do.

Performance Evaluation Since we have not yet conducted a performance evaluation, we cannot claim to have solved the slow compilation of Stratego code in Spoofax projects.

Deletion We need to handle deletion on multiple levels. Currently we only handle the deletion of a strategy, which requires the deletion of the corresponding intermediate file for that strategy. This is done by the front-end task. First all generated files that match the module name are removed. This can be done without needing to remember which ones were output the last time since the output file names follow a predictable, disjoint pattern. Then the front-end generates each file anew as necessary.

File level deletion is a similar problem to deleting a strategy, but now a front-end task doesn't get reached. This should be a trigger to still run the old task, with a special signal of deletion, to allow the task to clean up generated files.

One could argue that deleting intermediate files is the responsibility of the incremental build system too. This is likely to be expensive, as the system would have to track intermediate file. But in the case of Pluto, this is already tracked as a dynamic sanity check that no hidden dependencies exist.

Automatic Task Dependencies Each back-end task depends on exactly those front-end tasks that generated the input files for it. These dependencies seem derivable, but this is not simple with dynamically discovered dependencies.

Acknowledgements

We would like to thank Gabriël Konat for explaining many things around incremental build systems.

References

[1] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015. A sound and optimal incremental build system with dynamic dependencies. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 89–106. https://doi.org/10.1145/2814270.2814316

[2] Eelco Visser. 2005. Transformations for Abstractions. In *5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005), 30 September - 1 October 2005, Budapest, Hungary.* IEEE Computer Society. https://doi.org/10.1109/SCAM.2005.26

[3] Eelco Visser and Zine-El-Abidine Benaissa. 1998. A core language for rewriting. *Electronic Notes in Theoretical Computer Science* 15 (1998), 422–441. https://doi.org/10.1016/S1571-0661(05)80027-1