Separate Compilation as a Separate Concern

A Framework for Language-Independent Selective Recompilation

Nathan Bruning
Separate Compilation as a Separate Concern

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Nathan Bruning
born in Dordrecht, the Netherlands

TU Delft
Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl
Abstract

Aspect-oriented programming allows developers to modularize cross-cutting concerns in software source code. Concerns are implemented as aspects, which can be re-used across projects. During compilation or at run-time, the cross-cutting aspects are “woven” into the base program code.

After weaving, the aspect code is scattered across and tangled with base code and code from other aspects. Many aspects may affect the code generated by a single source module. It is difficult to predict which dependencies exist between base code modules and aspect modules. This language-specific information is, however, crucial for the development of a compiler that supports selective recompilation or incremental compilation.

We propose a reusable, language-independent framework that aspect-oriented language developers can use to automatically detect and track dependencies, transparently enabling selective recompilation. Our implementation is based on Stratego/XT, a framework for developing transformation systems. By using simple and concise rewrite rules, it is very suitable for developing domain-specific languages.
Preface

This research started out with the seemingly simple question whether we could modify the WebDSL compiler to operate faster. The famous XKCD phrase “I’m not slacking off, my code’s compiling”\(^1\) was regularly used in the department to accommodate a large number of coffee breaks. Even though, developers got annoyed by the unproductivity of waiting for compilation.

As always, the simple research question proved to be far more complicated to answer than initially expected. Traditional solutions do not work well with the WebDSL compiler, mainly because of the aspect-oriented nature of the language. The compiler itself is in continuous development, fixing just as many bugs as are introduced. More importantly, we gradually and ambitiously broadened the research question, eventually creating a language-independent framework to allow separate compilation for any compiler.

After an introductionary seminar course, the research started in the fall of 2009. Quickly, we pin-pointed some issues in the compiler implementation causing compilation times to explode for large projects. Applying caching techniques led to greatly decreased compilation times. Halfway 2010, the “low hanging fruit” had been consumed, and we started thinking about a more generally applicable solution to our problem. Due to different causes, the project slowed down significantly, coming almost to a full stop at multiple points in time. I would like to thank all that have convinced me to finish this research. I’ve had to pay quite a few rounds due to lost bets on my graduation date. I like to thank Eelco Visser for continuing support, and Danny Groenewegen for helping me, both with starting up the project and with finishing it. Finally, thanks to the fellow researchers in the department for providing me with a seat in their office, their knowledge and their company.

Nathan Bruning
Delft, the Netherlands
August 14, 2013

\(^1\)http://xkcd.com/303/
# Contents

Preface iii  
Contents v  
List of Figures vii  

1 Introduction 1  
1.1 Separation of Concerns 2  
1.2 Software Compilation and Execution 2  
1.3 Incremental Compilation 4  

2 Separate Compilation and Aspect-oriented Languages 5  
2.1 Aspect-oriented Language Features 6  
2.2 Modular Reasoning and Obliviousness 6  
2.3 Implementations 8  
2.4 Problem Statement 9  

3 Algorithms for Selective Recompilation 11  
3.1 Compilation Model 11  
3.2 Selective Recompilation 14  
3.3 Selection Criteria 15  
3.4 Existing Techniques 17  
3.5 Typechecking Algorithms 24  
3.6 Transformation Algorithms 31  

4 The NWL language 41  
4.1 Introduction 41  
4.2 The Language 42  
4.3 Implementation 44  

5 Language Independent Library 51
## CONTENTS

5.1 Stratego/XT and the Compilation Model .......................... 51
5.2 Information Store Traceability ........................................ 53
5.3 Library Implementation ................................................. 56
5.4 Testing ................................................................. 65

6 Case study: WebDSL ......................................................... 69
  6.1 The WebDSL language .................................................. 70
  6.2 Compiler Implementation .............................................. 71
  6.3 Pipeline Separation ..................................................... 76
  6.4 Development Environment ........................................... 87
  6.5 Library Integration ................................................... 97

7 Related Work ............................................................... 101
  7.1 Dynamic weaving approaches ....................................... 101
  7.2 Observers and Assistants ............................................ 104

8 Conclusions and Future Work ........................................... 107
  8.1 Contributions .......................................................... 107
  8.2 Conclusions ............................................................ 108
  8.3 Future work ............................................................. 108

9 List of Software Contributions ........................................ 111

Bibliography ................................................................. 113
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Whole-program compilation</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Separate compilation</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>Selective recompilation</td>
<td>13</td>
</tr>
<tr>
<td>3.4</td>
<td>Example of two Java classes in which removing the declaration-site for a method call does not invalidate the call-site</td>
<td>16</td>
</tr>
<tr>
<td>3.5</td>
<td>Example of two Java classes in which changing a local variable invalidates the call-site of the function containing local variable</td>
<td>17</td>
</tr>
<tr>
<td>3.6</td>
<td>Compilation of a C program</td>
<td>18</td>
</tr>
<tr>
<td>3.7</td>
<td>Using header files</td>
<td>19</td>
</tr>
<tr>
<td>3.8</td>
<td>A sample Makefile for a program consisting of two C source files</td>
<td>20</td>
</tr>
<tr>
<td>3.9</td>
<td>A Java file importing another class</td>
<td>22</td>
</tr>
<tr>
<td>3.10</td>
<td>Dependency analysis as performed by javac from Oracle compiler</td>
<td>23</td>
</tr>
<tr>
<td>3.11</td>
<td>A source definition describing a Java class</td>
<td>25</td>
</tr>
<tr>
<td>3.12</td>
<td>Three AST’s during the model-to-model transformation stage</td>
<td>32</td>
</tr>
<tr>
<td>3.13</td>
<td>Dependency graph at definition level</td>
<td>35</td>
</tr>
<tr>
<td>3.14</td>
<td>Context information creation and usage during three transformations</td>
<td>37</td>
</tr>
<tr>
<td>3.15</td>
<td>Dependency graph with information store keys</td>
<td>37</td>
</tr>
<tr>
<td>3.16</td>
<td>List of information transactions</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Simple NWL program source</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Extend entity in NWL</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>The AST of a sample NWL module</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Rule creation in Stratego</td>
<td>52</td>
</tr>
<tr>
<td>5.2</td>
<td>Parsing of modules in the AST</td>
<td>57</td>
</tr>
<tr>
<td>5.3</td>
<td>Sample ATerm stamp file</td>
<td>58</td>
</tr>
<tr>
<td>6.1</td>
<td>WebDSL compiler speed for various project sizes</td>
<td>70</td>
</tr>
<tr>
<td>6.2</td>
<td>WebDSL compiler pipeline</td>
<td>72</td>
</tr>
<tr>
<td>6.3</td>
<td>Attribution of time spent by the WebDSL compiler in the front- and back end for projects of various sizes</td>
<td>78</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.4</td>
<td>Cached generation: the creation of a cache file for a definition</td>
<td>81</td>
</tr>
<tr>
<td>6.5</td>
<td>Cached generation: the usage of a cache file for a definition</td>
<td>81</td>
</tr>
<tr>
<td>6.6</td>
<td>WebDSL compiler information flow</td>
<td>84</td>
</tr>
<tr>
<td>6.7</td>
<td>WebDSL compiler information flow, old-new comparison</td>
<td>86</td>
</tr>
<tr>
<td>6.8</td>
<td>Benchmarks of back end caching using a “hello-world” application</td>
<td>87</td>
</tr>
<tr>
<td>6.9</td>
<td>Benchmarks of back end caching using a medium-sized application</td>
<td>88</td>
</tr>
<tr>
<td>6.10</td>
<td>Typecheck rule for a Spoofax-based IDE</td>
<td>89</td>
</tr>
<tr>
<td>6.11</td>
<td>Sharing code between WebDSL compiler and Spoofax-based IDE</td>
<td>90</td>
</tr>
<tr>
<td>6.12</td>
<td>Eclipse IDE showing feedback for a failed consistency check</td>
<td>90</td>
</tr>
<tr>
<td>6.13</td>
<td>Subset of the call graphs used to determine which dynamic rules are relevant for the editor typechecker</td>
<td>94</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Software has become increasingly large and complex over time. In the early days of programming, a software program consisted of a linearly executed sequence of instructions. This imperative programming paradigm was augmented with the concepts of objects and classes in the early ’60s. In the past few decades, many more programming styles have been introduced: functional, declarative, event-driven, generic, modular and aspect-oriented programming (AOP), to name a few.

Aspect-oriented programming is a relatively new programming paradigm, introducing aspects. An aspect is comparable to a module, but while a module tries to isolate a specific part of functionality of the software project, an aspect isolates behavior applicable to multiple parts of the system. This introduces separation of concerns: the implementational description of a specific concern is confined to one aspect, even though the functionality described in the concern cross-cuts other parts of the software system.

Aspect-oriented languages have been discussed and assessed in various studies. Some of these language, like the popular AspectJ, have been used commercially for large software projects.

While the performance of computer hardware has vastly increased over time, so has the size and complexity of software programs. Typically, the workflow for a software developer includes the implementation of new functionality, repairing existing implementations and testing. Between implementing and testing the software, another step is often present in their workflow: compiling or building the software. This step is often an automated step, not requiring any effort from the developer; however, this step can take up a considerable amount of time, depending on the performance of the build system, the speed of the hardware used and the size and complexity of the software project. Incremental compilation aims to reduce the time spent in this step by optimizing the build procedure.

Because of the nature of aspects, incremental compilation techniques that are currently applied to compilers do not translate easily to a compiler for an aspect-oriented language. This work describes new techniques for incremental compilation applicable to compilers for these languages.
1. **Introduction**

1.1 **Separation of Concerns**

Dijkstra introduces “separation of concerns” [17] as a methodology for modularizing software systems. The term refers to the fact that a single program often has different concerns or tasks; for example, many programs have a pipeline of reading input, performing computation and writing output. This can be seen as three concerns or “modules”. Many different schemes of separating concerns can be used.

We can identify a number of benefits that are gained by using separation of concerns in a large software system. From a programmers perspective, overview of the source code is quickly lost as the size of a program increases. By splitting up a large and multi-functional systems in (largely independent) parts, the individual modules are easier to grasp for a developer. The division of a software system in modules also allows a developer to isolate certain functionality from the main program. If sufficiently decoupled, modules can be reused across software projects.

Separation of concerns also has managerial benefits [46]. The complete set of modules of a system define the functionality of such a system. Each module has a specific task which can be defined separately. Modules often need to interact; if not, we could consider the system as consisting of multiple independent systems. The methods using which one module interacts with another, is called the **interface** between two modules. Many design methodologies are based on keeping this interface relatively small. Each function of a module can be described by a black box: given a certain input, the module should perform a certain operation, optionally yielding some output. By formalizing the interface before actually implementing the modules, developer teams can independently build modules. This provides parallelism of development, decreasing the overall development time of a software project.

With the advent of the object-oriented programming paradigm, classes naturally became the granularity of module separation. This effectively translated the issue of how to divide modules into the question of how to identify classes in the design phase. Providing in the increased need for decomposition, various programming languages introduced the concept of packages, creating a multi-level hierarchy of decompositions.

1.2 **Software Compilation and Execution**

Regardless of the programming style of a specific language, computer programs are written to perform a certain function. The **execution** of a computer program happens within its execution environment; such an environment can consist of a hardware platform, an operating system, a “virtual machine”, mechanisms for data input and output, and so on.

Many programming languages provide a compiler. A compiler is computer program on its own. Its input consists of a computer program in the compiler’s **source language**, which the compiler translates to another computer program written in the **target language**, called the **compiled** program. The semantics of both programs is equivalent. Early compilers included the **assembler**, which translated human-readable processor instructions to machine-decodable bit sequences. The input language for an assembler has a syntax consisting of operations (processor instructions), operands (registers, memory locations and such) and
1.2. Software Compilation and Execution

special characters that determine the lexical structure (whitespace, line breaks). The output language is a binary format defined by the processor architecture used. Therefore, for each processor architecture a different compiler is required.

Other programming languages do not provide a compiler, but rather introduce an interpreter for software programs written in the language. An interpreter takes a software program in the source language, just as a compiler does. However, the interpreter does not provide another program as output. Having read the input program, the interpreter actually performs the instructions or computations that the input program describes. In other words, the interpreter provides an execution environment for the input program and executes the program in this environment.

Avoiding an ongoing discussion whether compilation or interpretation is “better”, we’ll merely discuss the main difference between the two techniques. Compilation is a one-time operation for a given source program. The compiler can perform optimizations while translating the program. As with all software, the compiler takes time to perform its operation. A compiler takes at least linear time to operate: the input program must be read and the output program must be written, so scaling more efficiently than linear is not possible.

On the contrary, an interpreter does not need time to read the whole input program. It does not need to analyze the whole input before translating and does not need to write any output program. Only the parts of the input program that are actually executed are strictly required to be read by the interpreter. The overhead of an interpreter depends on the speed with which the interpreter can execute the instructions provided by the input program. For each source instruction encountered, the interpreter must decide how to execute this instruction, given the execution environment in which the interpreter itself operates. Following the example of an assembler, an assembly interpreter must translate each human-readable processor instruction into an executable processor instruction on the fly.

With respect to the performance of the two execution models, a compiler has to perform a one-time operation on each modification of the input program. This compiled program is ready to be executed in its execution environment without further intervention of the compiler. For an interpreter, there is no initial delay caused by the compiler; the program can be run directly. However, for each execution of the interpreted program, there is a performance penalty tied to the on-the-fly translation of the instructions.

Many optimizations have been applied to both execution models. Compilers have access to the input program as a whole, and are therefore able to perform optimizations not only based on local context information, but can also perform optimizations that require whole-program analysis. On the other hand, a compiler is invoked before the program is actually ran, and therefore has no knowledge of dynamic properties of the program; how often is a specific piece of the program run during an execution, what are the values of the input variables to a program? An interpreter is able to keep track of all kinds of dynamic runtime properties and uses this information for optimizations.
1.3 Incremental Compilation

The workflow of a typical developer using a compiled programming language typically consists of:

- working on one or more source files,
- compiling the source program
- deploying the compiled software program to the execution environment
- executing the program
- testing whether the program executes correctly.

These steps in the developer workflow are applicable for traditional development as well as test-driven development. For larger programs, a single developer will not modify all modules of the system. For even larger programs, a single iteration of the developer workflow will modify only a small part of a module. Larger changes are often performed in multiple smaller steps, because small changes are easier to test.

We therefore assume that developers typically modify only a small part of the software system under development, before compiling the program and testing it. Two subsequent compiler invocations often process a lot of input files that have not changed between the two invocations.

A compiler translates an input program to a program in the target language, which is stored on some (typically non-volatile) medium. When invoking the compiler for a second time, the output files resulting from the first compiler run are still present on this medium.

These observations introduce a new optimization technique that greatly reduces compilation time: incremental compilation. After an initial compilation invocation, the compiler reuses (part of) the previously computed output.

The next chapter describes what techniques are being used to implement incremental compilers. Also, we’ll discuss why most of these techniques are not usable for creating an incremental compiler for aspect-oriented languages.
Chapter 2

Separate Compilation and Aspect-oriented Languages

For many purposes, the traditional module decomposition works well because the resulting layout of modules is logical from both a functional and an implementational view. Some concerns, however, do not have a clear destination in the system decomposition, because they provide “cross-cutting functionality” and influence large parts of, or even all of the system. These concerns have been introduced as cross-cutting concerns. Examples of these concerns include logging, tracing, profiling, billing and debugging [34].

Aspect-oriented programming introduces a new programming paradigm: aspects. An aspect is comparable to a module, but isolates behavior applicable to multiple parts of the system. Such an aspect is then horizontally ejected from the class hierarchy, handling it as a first-class element [18]. For example, a logging aspect defines that a certain message is printed on entry of each method; the scope of such an aspect would be all methods in the system, thereby cross-cutting all modules. Besides the concerns already mentioned, less trivial concerns have been identified as aspects: contract enforcement, runtime configuration management and change monitoring [34], for example. More detailed, multi-dimensional classifications of aspects have been proposed as well [35].

An early evaluation of aspect-oriented programming has empirically compared development in languages with and without aspect-oriented features [58]. The evaluation, while performed used a small group of test subjects, yielded some interesting results. The experiments revolved around the test group finding bugs in a software program. This software program was implemented in an aspect-oriented language as well as a traditional language. Subjects working with the aspect-oriented program generally located bugs quicker, if the bugs manifested themselves in aspect code. Moreover, the researchers conclude that aspect-oriented programming is easier to grasp for developers, as long as the base language and the aspect language are sufficiently decoupled.
2. SEPARATE COMPILATION AND ASPECT-ORIENTED LANGUAGES

2.1 Aspect-oriented Language Features

Aspect-oriented languages are programming languages that use the “aspect-oriented” programming paradigm. Many languages combine this paradigm with more traditional paradigms, like object-oriented programming or imperative programming. AspectJ, for example, adds aspect-oriented features to the object-oriented Java language [33].

While the syntax and usage of aspect-oriented languages may vary with the underlying paradigms, some features are common to all aspect-oriented languages.

Definition 1. Aspect: an aspect describes a particular function or contract of a software system. In contrast to a traditional module, an aspect can be cross-cutting, spreading over multiple components of a module decomposition. An aspect is described by its two properties:

1. Advice: the functional description of the aspect; which semantics does the aspect add to the system?
2. Pointcut: the impact of the aspect; which components of the system are influenced in behavior by this aspect? A pointcut is a set of join points: points in the execution flow of the program.

In AspectJ, an extension to the Java language, advice is represented similar to a function: a list of statements to be executed sequentially. A pointcut must be written as a boolean expression using various conditions; a (wildcard) match for a specific function name, and matches on the (formal) parameters of a function. Each join point can be qualified to specify whether the advice has to be executed before or after the matched points in the execution flow.

2.2 Modular Reasoning and Obliviousness

Aspect-oriented programming is often motivated by referencing to three of the expected benefits of choosing a good system decomposition, as specified by Parnas [46]:

1. Managerial benefits: because of parallel development on multiple component, implementation time is decreased.
2. Flexibility: components can change independently (as long as their interface to other components does not change).
3. Comprehensibility: one can look at the components of a system independently, which is easier than trying to grasp a large system as a whole.

The first benefit states that a decomposition should have managerial benefits to the developers, enabling separate development of modules. Secondly, the system as a whole is expected to be more flexible because (drastic) changes to one part of the system do not affect other parts, as long as the interface and specification remains the same.
2.2. Modular Reasoning and Obliviousness

Research of aspect-oriented programming mainly focuses on these two criteria and programming paradigms are evaluated by these criteria. For aspect-oriented languages, the first two of three criteria for decomposition of software systems are fulfilled by enabling the isolation and extraction of aspect code [14]. However, Parnas proposes a third benefit: the comprehensibility of the system as a whole should increase by decomposing the system. Trying to grasp both the connections between modules and the implementational details of all modules in large systems is generally considered impossible for humans. Separating functionality in modules should enable a developer to pick one of two perspectives: high-level or low level. At a high-level perspective, only the connections between modules remain visible. At a low-level, separate modules can be understood independent of the others. The low-level perspective allows for modular reasoning; assertions can be made based on the source code of individual modules. For example, in many object-oriented languages, a class can be seen as an individual module describing its functionality. However, code in a class is allowed to call methods on objects of another type. This introduces a dependency: when the implementation of the referenced class changes, the functionality of the referencing class changes as well. In many object-oriented languages, referenced types are explicitly imported into the namespace of the referencing class. This introduces a graph of type usages, where each import implies an edge in the graph. If any two classes are not connected (there is no path between them in the graph), their functionality is completely decoupled.

For languages with aspect-oriented features, the coupling between code and the applied advice is often implicit. The code on which advice applies (the pointcut of the aspect) has no explicit reference to the advice code. This is referred to as obliviousness by Filman and Friedman [20]. It is a core concept for aspect-oriented languages: the places in code which “receive advice” do not have to be specifically prepared for this reception. Note that when these places would be specifically prepared for reception of advice, the aspect references would be scattered through the code, disallowing for any separation of concerns. Filman and Friedman state that obliviousness is one of two properties that distinguish aspect-oriented languages from other languages, and reason that “better AOP systems are more oblivious”.

Aspect-oriented programming decouples cross-cutting aspects from the rest of the system (the components); however, these aspects do still influence the behavior of modules. Therefore, in aspect-oriented programming languages, modules by themselves can no longer be understood as being independent functional units. Aspects from anywhere in the system might influence the behavior of a module. The link between a module and its matching aspect behavior is not explicit in the source and can be determined at compile time or runtime, depending on the aspect weaving mechanism. Because static reasoning about the behavior of one individual module is no longer possible, the advantages of modular reasoning are lost.

For example, aspects in AspectJ are weaved in the modules at compile-time, described in detail in [27]. The modules and methods to which advice is applied are determined by matching the pointcut of an aspect to the available join points. Opposed to the normal Java source, a module does not have any reference to the advice that is to be applied to that module. By looking just at one module, it is not possible to reason which aspects are applied to it. Whole-program analysis is needed which considers the target modules, the
modules it depends on and all aspects. Modular reasoning is therefore impossible with AspectJ systems.

2.3 Implementations

Quite a few aspect-oriented languages exist today. AspectJ [33, 34] is perhaps the most commonly known. Others include Hyper/J [45], MultiJava [13] and AspectCOOL [5]. While all of these languages have aspect-oriented features, each has some distinguishing characteristics [20].

Whilst these languages differ in syntax and expressivity, the main concepts (aspects, advice, join points, pointcuts) apply to all of them. More interesting are the different implementational approaches that the various language designers have taken. From a compilation perspective, an important choice is at which point in time the advice is matched to the pointcuts. This matching is called aspect weaving; advice is inserted on the join points in target modules, as specified by a pointcut. Implementations can be divided in two main groups; one using static weaving, and one using dynamic weaving techniques. Static weaving occurs at compile-time, while dynamic weaving happens at load- or run-time.

Subdividing the static approaches, weaving can be seen as a source-to-source (model-to-model) transformation or as a compiler post-process step. The first approach has been taken by the WebDSL compiler [26], which uses aspect-like definitions to isolate concerns like access control. Most aspect-oriented languages, however, use the second approach, which takes the target code and weaves in the aspects. Hyper/J [45] popularized this approach, taking the bytecode output of a standard Java compiler as input for aspect weaving. It was also AspectJ’s initial weaving mechanism [27], although later versions have support for dynamic weaving as well. One particular argument for using bytecode as the source for aspect weaving (instead of source code) stems from [59]: the authors explain the benefits of applying aspects to off-the-shelf Java applications, for which often no source code is provided.

Dynamic weaving requires a mechanism to weave in the aspects after compilation. Existing approaches can be classified in three categories. First, load-time weaving (supported in modern AspectJ versions, AspectWerkz [8] and more) must be executed each time a new module is loaded in the system. Particularly, for AspectJ and AspectWerkz (both based on Java) the Java classloader is replaced to provide weaving functionality. Secondly, for an interpreted language, the virtual machine can be extended to support dynamic aspect weaving. The Jikes research virtual machine for Java has been extended so that it looks up aspects when execution hits a join point [47]. Steamloom [7] is another virtual machine implementation supporting aspects, doing bytecode transformations at runtime. A third category of dynamic implementations provides native support for aspects. In such an architecture, the front end of the compiler compiles the aspect-oriented language to a partially woven model containing separate meta-data. The front end performs expensive non-local optimizations, while the virtual machine uses the meta-data to do local optimizations. The meta-model is used in the virtual machine to dispatch aspect-oriented code, without actually weaving the code at bytecode level.
2.4 Problem Statement

With static weaving approaches, cross-cutting aspects are separated at source level only. After the phase of advice weaving in the compilation pipeline, aspects appear scattered throughout the target code. While isolating and lifting the aspect from the system at source level, in the compiled (target) code the opposite happens: the advice code becomes tangled with the individual modules it affects. This has several disadvantages. Following the same reasoning as we did for the source-code level, modular reasoning at target-language level is made impossible by using aspects. This affects for example unit testing, because the definition of “unit” now differs in pre- and postcompilation stages [48]. Source-level debugging becomes unintuitive when the line of execution for a program is a continuous back-and-forth between base code and aspect code.

Moreover, the scattering of source code in the compiled program impedes the possibility for separate or incremental compilation. Alongside the non-obvious dependencies between source files and target files introduced by weaving cross-cutting concerns, the obliviousness of aspect-oriented languages leverages the problem to a new level: additional dependencies are introduced by join points. The absence of an explicit link between advice and the code it advises to is problematic. Advice-to-component dependencies requires expensive static analysis at least; dynamic join points nullifies the compiler’s knowledge of dependencies between different source files, and dependencies between source files and compilation units.

Static weaving is used in many implementations because of the minimal runtime performance impact of using aspects. The pointcuts are matched statically as much as possible, eliminating the execution of any matching code at runtime for join points at which no advice matches. Note that many pointcuts (especially these including dynamic tests and control-flow criteria) still require runtime tests. More recently, dynamic matching has been getting more popular, for multiple reasons. In the absence of static weaving, compilation of a module is not dependent on any aspects, enabling true separate compilation. However, dynamic weaving comes at the cost of runtime overhead, which is not always acceptable. Various benchmarks have been performed. Overhead has been measured to be 37% for a particular implementation [1]. Comparing many AOP implementations, overhead introduced by dynamic weaving varies from 12% to 35% [53].

At least, compiler implementations using load-time or dynamic weaving have an intrinsic runtime overhead that is not always acceptable. In contrast, static weaving requires very complicated dependency analysis by the language developer.

This thesis will discuss the following question:

*Can separate compilation for a statically-weaved aspect-oriented language be achieved by creating a library for language-independent automatic dependency analysis?*
Chapter 3

Algorithms for Selective Recompilation

In this chapter we will define an algorithmic solution to the problem of selective recompilation for a model of a generalized compiler. We will show that this compiler model is applicable to many compiler implementations, including aspect-oriented compilers.

3.1 Compilation Model

A compiler transforms or translates a program from a source language to a target language. The target language can be a set of machine instructions or another human-readable language.

As compilation involves translating the source program, the execution time and memory requirements of a compiler scale linearly in the size of the input program, at least.

While developing, the compiler is typically invoked multiple times, each input being an updated version of the last input. When only minor changes have been made, much of the output in the target language is still valid; recompiling unchanged input files will often result in the same output file being generated. To take advantage of the fact that sequential compilation runs typically share a large part of their input, incremental compilation techniques are typically used to increase compiler performance. We classify three types of compilers with respect to the incremental compilation technique used:

1. Whole-program compiler: translates the entire input on each invocation
2. Separate compiler: is able to compile input files one at a time
3. Selective compiler: performs compilation only on input files that need recompilation

A whole-program compiler takes a full program as input, typically spanning multiple input file. Its output consists of a complete program in the target language. This technique is illustrated in figure 3.1.

A compiler utilizing separate compilation compiles one input file on each invocation. The responsibility of deciding which input files have to be considered is left to the user, or
to another tool. Whether separate compilation is possible depends on the input language; for example, in the presence of cross-cutting aspects in an aspect-oriented language, the compiler needs knowledge of all join points before translating an input file. Therefore, an aspect-oriented compiler with static weaving is not eligible for true separate compilation. A schematic example of separate compilation is shown in figure 3.2.

With selective recompilation, the compiler decides which of the input files have to be considered. For unmodified files, the output files generated by a previous compiler invocation are considered to be up-to-date; they do not have to be recompiled. This is shown in figure 3.3. The compiler starts by checking which source files have been modified since last recompilation. These files have to be recompiled. By using dependency analysis, the compiler can mark other input files for recompilation during the compilation pass. The concept of selective recompilation is elaborated upon in section 3.2.
3.1. Compilation Model

![Diagram of selective recompilation](image)

**Figure 3.3: Selective recompilation**

3.1.1 Definitions

We define a “top-level definition”, or simply a “definition” as a unit of compilation. Depending on the source language, a definition can span a file, a module, a package, a class, and so forth.

For each recompilation, a definition can be either unmodified (compared to its contents during a previous compilation) or modified. For this section, it suffices to pick a file as the granularity of a top-level definition. This eases the detection of changes to definitions, as operating systems keep track of modification time on a per-file basis.

The compilation of a program can be defined as the transformation of a set of source code definitions \( S_{i,j,...} \) to a set of target code definitions \( T_{i,j,...} \). Note that a one-to-one correspondence between source and target definitions is not a necessity, as one source definition can generate multiple target definitions. Because of aspect weaving, the existence and the contents of a target definition can be dependent on any number of source modules. Therefore, the mapping between source and target definitions is, in general, a many-to-many mapping.

For some languages however, the mapping can be simpler. For example, a C source file compiles into exactly one object file, resulting in a one-to-one mapping. Java source file contain one class, which is compiled into one binary class file; with the exception that a class in a source file can contain inner classes, which are compiled into their own binary class files. For Java, therefore, the mapping is not necessarily bijective but always surjective\(^1\).

3.1.2 Context Information

For all non-trivial compilers, the translation of source level constructs is dependent on the context of the construct. For example, the declared type of a variable used in an equation is relevant for typechecking; for example, certain arithmetic operations might not be allowed on operands of mismatching types. This type information can also be relevant for the target

\(^1\)However, the mapping is not bidirectional, as decompilation does not guarantee an exact representation of the source code.
language translation of the computation; different instructions can be chosen for integer or floating point arithmetic, for example.

Such type information is in many languages not contained within the arithmetic statement, but determined by the declaration site of the used variables. Also, types are often inferred by language rules; the result type of the addition of two integers is again an integer, a floating point number added with an integer results in a floating point number, and so forth.

For a compiler to “remember” the types of variables, the compiler needs a memory that is global to the compilation of the program, that is, information gathered by examining one part of the program can be used for translating another part of the program.

As a generalization for all compiler implementations, we introduce the notion of an information store. It represents a data store that the compiler can use as global memory during compilation. For many compilers, this information store is generally referred to as the symbol table. Our information store, however, is not confined to type information about symbols.

“Information” can be added, removed, modified and retrieved from this store. Information in the store is modeled as a map containing key-value pairs: for each key, a list of values can be stored. Our algorithm implementation expects the store to be traceable: access and modification can be monitored during execution of the compiler pipeline.

The information store is defined by the following operations:

1. **READ(key)**: retrieves the set of values stored for the given key
2. **ADD(key, value)**: add the value to the set of values for the given key
3. **REMOVE(key, value)**: remove the value from the set of values for the given key

### 3.2 Selective Recompilation

A compiler employing selective recompilation “selects” which of the source definitions need to be recompiled. During the initial compilation of a program, all sources have to be compiled. During each following compilation invocation, the compiler uses an algorithm to determine which source definitions to recompile, and for which source definition the translated output can be taken from the previous invocation (i.e. the output is already present on disk).

#### 3.2.1 Caching and Memoization

Definitions are *independent* if the translation (output) of the compiler for a given source definition does not depend on any other source definition. For a language in which definitions are independent, we can define a function $\text{COMPILE}(S_i) = T_i$ for each definition $S_i$. The compiler is the implementation of this function. For independent definitions, selective recompilation is trivial to achieve. Only these definitions that have been modified since the last compilation run have to be recompiled.
Memoization is the technique of remembering (storing) answers for a function to avoid recalculation [42]. For a deterministic function, the outcome of a function application is determined solely by the function itself and its input. The memoization technique is applicable for compilers by using caching. The compilation of a definition results in a target definition, typically written to disk. Recompiling the same definition results in the same target definition, rendering the re-computation and disk writing useless, as the output file is already present.

However, because the definitions together form a software program, they often have interdependencies. For example, if a function defined in definition \( S_i \) uses a type defined in \( S_j \), we have a dependency \( S_i \rightarrow S_j \). For typechecking purposes, \( S_i \) is dependent on the signature of \( S_j \). If inlining is used, also the implementation of the type defined in \( S_j \) is input for the compilation of \( S_i \).

### 3.3 Selection Criteria

For a compiler to provide selective recompilation, it must be able to determine the impact of a change in a source definition for the compiled language. The impact of a change is the set of definitions for which the translation by the compiler is influenced by that change; i.e. of which the associated output definition(s) change.

The impact of a change results from the semantics of the language constructs of the source language. There are two ways to specify this impact:

1. **Forward dependencies**: given a change in a definition, which (other) definition will be impacted during their translation?

2. **Reverse dependencies**: given a definition, which changes to other definitions will have impact on the translation of this definition?

Some languages are designed with separate compilation in mind. For example, in Java the backwards dependencies of a class (file) are summed up above the class declaration of the source file: import statements describe which other classes are used. This describes a set of dependencies: the class depends on all of its imports. For example, a simple case is when an imported class is removed; the file importing the now removed import is no longer valid; a typechecking error must be presented.

In order to support selective recompilation, the designer of a compiler must set up a list of rules that specify when a change influences other definitions: the forward dependencies. For Java, some observations can be easily made:

1. Changing the signature of a function must cause re-evaluation of the typecheck rules to all the call sites of that function.

2. Changing the name of a local variable does not impact the translation of other definition. Therefore, this change does not need to cause recompilation of other classes, even if an import has been declared.

---

2Section 3.4.2 describes selective recompilation for Java in more detail.
3. Changing the superclass of a class requires the recompilation of all subclasses.

We argue that such predicates are very hard to construct for programming languages, in particular for aspect-oriented languages. Being too selective in the list of recompilation rules results in definitions not being recompiled when they need to be; inconsistent or invalid compiled programs can arise. Fearing inconsistency by small optimizations in the recompilation rules, language and compiler developers choose either to provide conservative but safe recompilation rules, or to provide no separate compilation at all.

Formal type systems for selective recompiations of subsets of Java have been proposed [3, 2]. Even with formal type systems and for subsets of the Java languages, it proves to be complex to identity all dependencies in the type system.

Furthermore, to indicate the difficulty of recompilation rules, let us consider the list of basic recompilation rules for Java we presented earlier. Rule 1 is based on the signature of a function. Changing the name of a function surely requires recompilation of call sites, as the calling statements are probably no longer valid; however, this is not guaranteed, as the changed function can have another overloaded variant. Listing 3.4 shows an example of such an scenario.

Rule 2 asserts that a local variable is indeed local, attaining the independence of that variable with respect to other definitions. However, consider an optimizing compiler performing inline expansion. When the function declaring the local variable is inlined in another definition, changes to that local variable now also affect the other definition, as shown in listing 3.5.

The third rule is a very conservative rule: indeed, in some cases the change of a superclass does change the compiler output of a subclass. But there are many cases as well for which there is no impact to subclasses. Whether there is any impact depends on the contents of the superclass, the contents of the subclass and the implementation of the compiler.

```
public class DeclarationSite {
    public void test(Object x) { }

    // Changing the name of this method does not
    // invalidate the CallSite class.
    public void test(String x) { }
}

public class CallSite {
    public void call(DeclarationSite param) {
        param.test("x b");
    }
}
```

Figure 3.4: Example of two Java classes in which removing the declaration-site for a method call does not invalidate the call-site
3.4 Existing Techniques

For high-level domain specific languages, code-generation often entails local-to-global and global-to-global transformations, that is, transformations which do not have a 1-on-1 mapping of source- and target code [52]. Aspect weaving, for example, is a local-to-global transformation, as an aspect may be applied in many places in the target code. Furthermore, the transitivity of dependencies causes inter-module dependencies that might not be obvious, even for the language developer.

We argue that language- or compiler developers should not be required to formalize the dependency rules of their language. It is too error-prone and dependencies have to be reconsidered at each language change. This chapter introduces a dependency tracking mechanism that automatically identifies situations in which compilation of module A is dependent on information provided by a module B.

3.4 Existing Techniques

Even in the absence of aspect-oriented features, incremental compilation or selective re-compilation are not easy to achieve. The techniques eligible for a compiler are greatly dependent on the source language of the compiler. Some functional programming languages, for example, allow true separate compilation; there are no inter-dependencies between different source files. However, for almost all software programs, source files refer to each other; by calling functions declared in another module, by using types defined in other modules, by inlining code fragments, and so on. Besides the options and restrictions provided by a source language, there can be multiple compiler implementations for the same language, differing in the approach to separate compilation. This section provides basic

```java
public class Declaration {
    public static int test() {
        int x = 3;
        return x + 1;
    }
}

public class CallSite {
    public int caller() {
        // The compiler performed inlining here; this was
        // return Declaration.test();
        int x = 3;
        return x + 1;
    }
}
```

Figure 3.5: Example of two Java classes in which changing a local variable invalidates the call-site of the function containing local variable
information on separate-compilation techniques used for programming languages that lack aspect-oriented features.

### 3.4.1 C and C++

The languages C and C++ date back to 1969 and 1979 respectively, and are rated the #1 and #3 most-used programming languages [40]. Many compiler implementations exist, most notably the GNU Compiler collection [50]. The GNU Compiler Collection (GCC) is a set of open-source compilers licensed under the GPL license. There are front ends for many languages, including C, C++, Java, Ada and more. The compilers have been adopted as the standard compiler for large software projects like Linux and BSD-products. All of the compilers share a back end, which can generate instructions for a large set of architectures, ranging from mainstream desktop processors to supercomputer architectures to embedded systems.

Independent of the compiler implementation used, the compilation of a C or C++ program proceeds in these steps, also shown in figure 3.6:

1. Preprocessing: the source files (usually with .c extension) are processed for special directives like includes and macros.

2. Compilation: the preprocessed source files are compiled into object files (suffixed with .o extension).

3. Linking: a number of object files are linked together to form one binary.

The first step reads the source files from disk and processes macros, includes and other special directives. The preprocessor takes source files and outputs translation units [28]. The compilation step translates the human-readable C files to object files: binary files containing machine instructions. Depending on the invocation of the compiler, the linking step will produce a stand-alone binary program (executable) or a library, to be included in other programs.

```plaintext
18
```

---

**Figure 3.6: Compilation of a C program**
In C, each entity (a variable, function, type) must be declared before usage. This implies an ordering in the source file. Sometimes, there is no ordering of declarations to fulfill the declare-before-use rule; for example, when two functions call each other. In this case, forward declaration has to be used. Before using an entity it has to be declared, but not necessarily be defined. Declaration defines the prototype of the entity. For a function, the prototype, or signature, consists of its name, parameter types and return type. For a variable, the signature consists of its name and type. For classes in C++, a prototype consists of the name and all its members or attributes. The definition is the actual implementation of the entity, that is, the statements of a function, the functions of a class and so forth.

The definition (implementation) for a forward declaration does not need to be present during compilation. Suppose file \( A \) contains \( f_a \), and file \( B \) contains \( f_b \) that calls (uses) \( f_a \). When file \( B \) includes a forward declaration of \( f_a \), the file \( B \) can be compiled in absence of file \( A \). The linker (which performs the linking step) can be instructed to verify the existence of \( f_a \). Note that this verification does not always need to be performed when linking: when using shared libraries, the execution environment (operating system) will load compiled libraries at runtime.

Using forward declaration, C provides a very basic form of separate compilation. A source file can be compiled separately, given that the signatures of the used entities are forward declared.

When an entity is used in many source files other than the source file containing the definition, this gets cumbersome. Each of these source files needs a forward declarations for the entity, duplicating source code over these files. Header files provide a mechanism to extract the forward declarations for a set of entities. Typically, a header file (with extension \( .h \)) exists for each source file. This header file contains only forward declarations. When a source file needs entities that are defined in another source file, it needs to include the header corresponding to the defining source file. Figure 3.7 depicts the use of headers for the files \( A \) and \( B \) described above. The arrow shows a use-def relation: file \( B \) uses an entity that is defined by file \( A \).

![Figure 3.7: Using header files](image)

Header files describe the signature of source files. When the implementation of an entity
changes, but not its signature, the header file is not modified. However, when the signature of an entity changes, for example when adding a formal parameter to a function, the files using this definition have to be recompiled. This introduces a dependency: source files are dependent on the contents of the included headers files, but not on the corresponding source files.

The GNU Make tool [49] provides a declarative language to specify how a software system has to be built, given the build commands and dependencies between source files. Figure 3.8 shows a *Makefile*, corresponding to the example with two source files *A* and *B*. Each of the three blocks describes one target, a single build step. A target consists of these elements:

- The filename before the colon determines the target: the resulting file of this build step.
- The filenames after the colon are the dependencies. If any of these files has been modified after the last time the build step was performed, the build step needs to be re-executed.
- The indented command line describes how to perform the build step.

The first target concretizes the linking step. Given two object files *a.o* and *b.o*, the invocation of *gcc* (the GCC compiler) will link these object files to produce an executable named *a.exe*.

The last two targets describe how to make a object file from a C source file. The command line is an invocation of *gcc*, the GCC compiler. Passing `-c` instructs the compiler to perform only step 1, the compilation, and to skip linking. The `-o` argument specifies the output file. For object file *a.o*, the dependencies consist of the source file *a.c* and also the header *a.h*, which is imported in *a.c*. For object file *b.o* however, also the header *a.h* is recited. This effects the dependency of source file *B* on the header file of *A*. When the signature of *A* changes, its header file *a.h* changes, and *make* will rebuild the object file of *B*.

```
 a.exe: a.o b.o
   gcc -o a.exe a.o b.o

 a.o: a.c a.h
   gcc -c -o a.o a.c

 b.o: b.c b.h a.h
   gcc -c -o b.o b.c
```

Figure 3.8: A sample Makefile for a program consisting of two C source files
3.4. Existing Techniques

Automake dependency handling

If a source file requires entities defined in another source file, it includes the header file providing forward declarations. These dependencies are made explicit in the source files; each include directive denotes a dependency on a list of entity signatures.

Even though these dependencies are already specified in the source files, they are repeated in the Makefile. This leads to duplication of information. Problems can arise when the headers included in a source file are not correctly represented in the Makefile. For example, consider a Makefile in which a header dependency is missing in the Makefile. The make tool will not trigger recompilation of the depending source file if the header file changes. This can lead to errors during linking or even worse, to errors at runtime.

Luckily, the GCC toolchain supports automatic dependency handling. When enabled, the GCC compiler will output which source files include which headers, thereby generating the dependencies. The automake tool automates the process of incorporating these dependencies in a Makefile.

3.4.2 Java

The Java programming language is an object-oriented language originally developed by Sun Microsystems. It follows the “write-once, run anywhere” principle; a compiled program can run on any computer architecture, provided there is a Java runtime present on the execution environment. Runtime implementations exist for most desktop operating systems, as well as many embedded operating systems.

In a Java program, each source file defines exactly one class. Classes have hierarchical names. The hierarchical parts are separated by a dot. The part of the full class name before the last dot is called the package; a container of classes and other packages. For example, when packages exist with the names com and com.example, a contained class is referred to as com.example.MyClass.

The input to a Java compiler is a source program in human-readable Java syntax. The output consists of a bytecode file for each class. The bytecode instructions are standardized by the language; the instructions are not tied to the instruction set of the execution environment, like the processor instructions.

Instead of using header files like with C or C++, Java computes signatures for required types automatically. A Java class can use other classes by importing these classes. Importing a class makes all type information of the imported class available in that file. For example, defining a local variable of a type needs the that type to be imported, as shown in listing 3.9.

Single classes can be imported by name, or a whole package (excluding its sub-packages) can be imported using a wildcard syntax: import com.example.*.

Some exceptions exist for the requirement of importing classes:

- A file can access all classes defined in the java.lang package, which contain standard classes of which the implementation is provided by the toolset. In other words, each Java file implicitly contains the wildcard import java.lang.*.

3 This is not true when using *inner classes*, which are defined inside the same input file.
3. Algorithms for Selective Recompilation

```java
package com.example;
import com.example.SecondClass;

class FirstClass {
    public void testMethod() {
        SecondClass var = new SecondClass();
        ...
    }
}
```

Figure 3.9: A Java file importing another class

- A file can access all classes in the same package. This amounts to the implicit import of `com.example.*` for the Java class `com.example.MyClass`.

- A file can use any class by providing its fully-qualified name, that is, its name prepended with the package name.

Oracle compiler

The Java compiler `javac` from Oracle (previously Sun Microsystems) has to be invoked by specifying a list of Java source files. Because there are no headers, the compiler is unable to perform semantics type checks without accessing the imported types as well. If any of these classes are not specified on the command line, the compiler will automatically take them in consideration while compiling. Figure 3.10 shows the decision graph for the Java compiler from Sun Microsystems when a unknown type is encountered during compilation [44].

This analysis is weak but safe. By checking whether the source file is newer than the bytecode file, the compiler notices changes in the implementations of used types. Thereby, semantic errors with respect to the changed imported types are caught. However, a change to a used type does not necessarily mean its signature has changed; only the signature is important for the typechecker.

Eclipse JDT

Another implementation of a Java compiler comes with the Eclipse platform [11]. Eclipse provides integrated development environments (IDEs) for various programming languages. Support for various programming languages is provided by plugins; the base platform of Eclipse is language agnostic.

The Eclipse JDT plugin / subproject [22] provides many components supporting the development of Java programs within Eclipse. The JDT core component provides a Java compiler called ECJ and an infrastructure for modeling Java programs in both source code and runtime environment.

22
3.4. Existing Techniques

Figure 3.10: Dependency analysis as performed by `javac` from Oracle compiler

Various tools of the JDT either use IBM software or originate from IBM projects. This is not a surprise given that the whole Eclipse project started as an IBM initiative. The ECJ compiler itself is based on IBM’s VisualAge compiler.

The ECJ compiler is not based on Oracle’s compiler and therefore both compilers have their flaws and features. The ECJ compiler is mostly celebrated for its ability to compile source files containing unresolved compilation errors, and its incremental compilation features.

The compilation of files containing errors (either syntax errors, semantic errors or other errors) is a feature often used in prototyping software. Sometimes parts of a software program can run independent of the availability or correctness of other components. For example, a Java class can have a method that has syntax errors; but as long as this method is not called at runtime, the operation of the program does not need to impacted. Many compilers will not generate any output for incorrect source files. The ECJ compiler allows the generation of binary Java files for source files containing errors. If at runtime such a compilation error is encountered (“reached”), an exception will be thrown, preventing continuation of normal program flow.

The incremental compilation feature of the ECJ compiler is much advertised but its techniques are not well documented, although open source. The ECJ compiler was based on Jikes, a Java compiler originally maintained as open-source project by IBM. ECJ still uses Jikes Parser Generated for generating a Java syntax parser. While Jikes, now detached from IBM, has not released any versions since early 2005, it’s website gives away some
details on the incremental compilation. Amongst other features, the Jikes compiler supports a continuous compilation mode. In this mode, after initial compilation, the compiler does not end but rather waits for user input. When the user finished editing files using its favorite editor, sending a key press to the compiler starts an incremental compilation: the compiler sets out to find the “minimum number of compilations needed to bring the class files into a complete and consistent state” [15]. Just as automake, the Jikes compiler provides an option to generate dependency information for inclusion in Makefiles.

3.5 Typechecking Algorithms

The typechecking stage of a compiler checks the static consistency of a program. These checks validate an input program with respect to the syntactic and semantic definition of the source language. Most importantly, this provides early feedback to the developer; for example, a compiler can detect statically when an unexisting function is called. Catching as many errors at compilation time is often preferred to using runtime checks, which are only shown when the faulty code is executed [25]. Another benefit is that ensuring “valid” input early in the compiler pipeline alleviates the need of thorough checks in the pipeline of the compiler following the typechecking stage.

3.5.1 Signatures

Traditionally, signature files are used to separate the interface of a definition from its implementational details. For example, for the typechecker to attain the semantic validity of a subroutine call, the compiler does not need to know the implementation of the subroutine; it suffices to known there exists a routine with a specific name that accepts the given parameter types, if any. Depending on the context, the return type of the function might be relevant as well. Therefore, the signature of a subroutine consists of its name, its formal parameter types and the return type.

Some languages use separate files for signatures, some languages include the signatures in the implementation files and others automatically derive signatures from implementation files. For example, section 3.4.1 describes how C uses header files to store signatures, and section 3.4.2 elaborates on the lack of signature files for the Java language.

While the signature technique has proved to be a sufficient solution for languages like C and Java, there are a few disadvantages. The language developer has to determine which elements of a language construct are relevant for typechecking purposes, and therefore have to be included in the signature file. Section 3.3 described the difficulty in finding a correct and minimized set of rules to determine the impact of changes. Similarly, the language developer is burdened by assuring that signature files contain all information relevant for typechecking. Having smaller signature files eases the job of the developer using the programming language. Therefore, the optimum for signature files is the smallest possible subset of the definitions contained in the related source file.

Besides the difficulty in finding the subset of information for signature files, in many cases the contents of the signature files is duplicated from the source definition. This leads to code duplication, as is the case for C headers.
3.5. Typechecking Algorithms

3.5.2 Consistency Rules

Our generic compilation model has two sequential stages that together perform typechecking of the input program:

1. Declaring
2. Constraint checking

The declare phase is a single pass over all input definitions in which the context information store is filled with information regarding the signatures of the definitions. During the second phase, each definition is checked for conformance to the consistency rules of the programming language. As an example, consider the following Java definition:

```java
class Test {
    public boolean f(int x) {
        return x < 5;
    }
}
```

Figure 3.11: A source definition describing a Java class

For a Java compiler, the typechecker could store the following \((\text{key} \rightarrow \text{value})\) pairs of information in the context information store:

- \((\text{CLASSES} \rightarrow [\text{Test}])\)
- \((\{\text{FUNCTIONS}, \text{Test}\} \rightarrow [f])\)
- \((\{\text{SIGNATURE}, \text{Test}, f\} \rightarrow [[[\text{int}], \text{boolean}]]))\)

In these information store entries, we’ve used \(\{\}\) to describe tuples and \([\]\) to describe an ordered list.

The first entry, with key \(\text{CLASSES}\), is used to store the fact that \(\text{Test}\) is the name of a class. For example, the compiler is able to check whether the superclass of a subclass exists using this entry.

The key of the second entry is a 2-tuple of \(\text{FUNCTIONS}\) and \(\text{Test}\). This key is the index in the context information store. The entry determines all of the functions that are defined within the class specified by the second element of the tuple. In the case of our code example, this would be a set containing \(f\) as only element.

The third entry is keyed with a 3-tuple: \(\text{SIGNATURE}\), the name of the class and the name of the function. The value associated with this key contains the typechecking signature of the function, in this case a tuple containing two elements. The first element is a list of all parameter types; the second element is the type of the return value. Note that for languages supporting method overloading, like Java, there is a list of signatures for each pair of class
3. **ALGORITHMS FOR SELECTIVE RECOMPIRATION**

and method name. In this case, however, only one signature is present for the combination of class and method name.

During the consistency check phase, these context information store entries can be used to determine whether a method call on an object of type `Test` is statically type-safe. Another consistency check would assert that there is only one class declared for each class name.

### 3.5.3 Automatic Signatures

Looking at the two phases of typechecking, one observation is that the first phase only writes to the context information store and the second phase only reads. It’s easy to see that a dependency between two definitions exists if information stored by one definition is used by another definition.

Formalizing this notion, for each definition $S_i$, let $\text{CREATE}_i$ define the set of key-value pairs that are stored during declaration of $S_i$. This set $\text{CREATE}_i$ entails the context information of $i$ that is available for other definitions to check their consistency; in other words, $\text{CREATE}_i$ is a representation of the signature of $S_i$. We call this the create set of a definition.

Typechecking can be described as a function $\text{TC}$ that results in a set of error messages describing which consistency checks have failed. These would typically be fed back to the user, on the command line or in an integrated development environment.

**Definition 2.** For a whole-program compiler that has inputs $S_{i, j, ...}$, the function $\text{TC}_{wp}$ can be described as:

$$\text{TC}_{wp} : S_{i, j, ...} \rightarrow \text{ERR}$$

where $\text{ERR}$ is the set of consistency violations.

An incremental typechecker does not need the whole program as input to check the consistency of a single definition. When a definition $S_i$ is first compiled, the set $\text{CREATE}_i$ is persistently stored in a cache (typically, on disk). Now when another compilation run is started, and the compiler detects that $S_i$ is not modified since the previous compilation, but $S_j$ has been modified, the compiler uses the cached $\text{CREATE}_i$ as input for the typechecking phase.

**Definition 3.** Let us postfix cached data with an apostrophe: the cached create set of a definition $S_i$ is denoted by $\text{CREATE}_i'$. Incremental typechecking of a program can be formalized using this function:

$$\text{TC}_{sig} : (S_i, \{\text{CREATE}_j' | j \neq i\}) \rightarrow (\text{ERR}_i, \text{CREATE}_i)$$

The function takes a source definition $S_i$ and a cached create set for all unmodified definitions. The result is a pair consisting of the set of consistency violations and the new create set for the modified definition.

By using the cached values $\text{CREATE}_i'$ for unmodified definitions, the compiler does not need to perform the declaration phase for any definitions that have not been modified. For large programs with only minor changes, this can save considerable time in the declaration
3.5. Typechecking Algorithms

phase. This optimization comes at the cost of reading and writing the create sets, and the disk space to store these sets.

There are only writes to the context information store during the declaration phase, and only reads during the constraint checking phase. Therefore, the information that a declaration exposes is exactly the create set generated during the declaration phase. By exposing information, a definition allows other definitions (in other source files) to use that information for typechecking purposes. Going back to the example in code listing 3.11, the signature of the method f of class Test is exposed through the information stored using the key containing SIGNATURE. The contents of the method, the comparison of variable x to the value 5 and returning the result of this comparison, is not exposed through the information store. No other source definition is aware of this comparison, and no definition can use this fact to perform constraint checking.

This split between exposed information and non-exposed information defines the automatic signature of a definition. The automatic signature of a definition is the create set of a signature; the information is sufficient for typechecking other definitions. Note that the compiler developer does not need to specify the contents of the create set; the contents are automatically determined using the tracking functionality of the context information store.

3.5.4 Dependency Handling

By using caching for the signatures, the compiler has to re-evaluate the declare phase only for modified definitions. However, even if a definition is unmodified, its set of consistency errors may change. Suppose a function f is removed from definition S_i and called from S_j; even if S_j is not modified, it now contains an inconsistency because of a call to a nonexistent function. A dependency mechanism is needed to trigger consistency checking of S_j in this case.

Dependencies emerge from the fact that one definition stores information (its signature) that is consequently used by another definition. We define a set DEP_{i\rightarrow j} of information store keys that were stored during declaration of S_i and used during constraint checking of S_j. This set of keys DEP_{i\rightarrow j} embodies shared data of S_i and S_j, introducing a dependency between the two definitions. First, as an abbreviation, we introduce the function K that extracts the set of keys for a create set (strips the values):

\[ K(CREATE_i) = \{ k \mid \exists (k, v) \in CREATE_i \} \]

Besides a create set, the compiler can also keep track of which information has been read from the context information store during consistency checks. For a definition S_i, the use set USE_i is the set of keys for which a lookup has been performed in the context information store during constraint checking of that definition.

**Definition 4.** The set DEP can be specified for each pair of definitions (i, j) as:

\[ DEP_{i\rightarrow j} = K(CREATE_i) \cap USE_j \]

This set contains all keys for which definition S_i has added a value to the information store, and S_j has performed a lookup of that key.
A global function defined in $S_i$ and called from $S_j$ is an example of such a dependency. For selective recompilation, an important observation is that a change in the values $\bar{v}$ for any key in $\text{DEP}_{i \rightarrow j}$ (caused by recompilation of $S_i$) invalidates the consistency of $S_j$.

Note that even if a lookup in the context information store does not succeed (i.e. the key has no associated value in the store), its lookup must be stored in the use set. For example, consider the following consistency rule: given a function, no function may exist with the same name. For a function $f$, this implies a lookup that checks for a function named $f$. Adding such a function to another definition in the program must trigger a revalidation of the duplicate function check.

Storing this use set for each definition alongside the create set, definition 3 of an incremental typechecker can be refined to the following function:

**Definition 5.**

$$TC : (S_i, \{\text{CREATE}'_j \mid j \neq i, \text{USE}'_j \mid j \neq i\}) \rightarrow (\text{ERR}_i, \text{CREATE}_i, \text{USE}_i)$$

The create and use set of definitions can be detected because our requirement on the context information store: it can be monitored, that is, any read or write operation can be detected.

With the formal definition of the function $TC$ in place, we can describe the implementation of $TC$. The following two methods differ in the granularity in which typechecking of dependent definitions is imposed. The methods will be exemplified with a scenario in which $S_a$ is to be compiled, and $S_{b,c}$ are unmodified and have cached use- and create sets.

**Method 1. Information-key Based Granularity**

A simple method for detecting dependencies is to intersect the use set with the keys of the create set, resulting in the set $\text{DEP}$ (definition 4). After declaring the modified definition $S_a$ and using the cached data from the last compilation run, the dependency information available to the compiler amounts to:

- The newly generated $\text{CREATE}_a$ set.
- The previously generated $\text{CREATE}'_a$ set.
- The previously generated $\text{CREATE}'_b$ and $\text{USE}'_b$ sets.
- The previously generated $\text{CREATE}'_c$ and $\text{USE}'_c$ sets.

Note that it is important to use both the newly computed $\text{CREATE}_a$ set and the cached $\text{CREATE}'_a$ set from disk. Consider these two cases:

- For a key $k$, an entry exists in the new set, but not in the old set. The definition has new information exposed, potentially invalidating the consistency of other definitions. For example, a global function name might now be declared in multiple places, which can be a consistency violation.
• For a key \(k\), an entry exists in the old set, but not in the new set. The definition previously has exposed information that is no longer present. For example, a global function was previously defined but has been removed. This affects other definitions, as their function calls may no longer be valid.

Given that the compiler has all relevant information read from disk, and considering the two cases described above, the following predicate can be used to determine whether re-evaluation by the typechecker is necessary for an unmodified definition (in case \(S_a\) is the only modified definition):

**Definition 6.** Re-evaluation of consistency checks for \(S_b\) is only necessary when:

\[
(K(\text{CREATE}_a \cup \text{CREATE}'_a)) \cap \text{USE}'_b \neq \emptyset
\]

or, equivalently:

\[
(K(\text{CREATE}_a) \cap \text{USE}'_b = \emptyset) \land (K(\text{CREATE}'_a) \cap \text{USE}_b = \emptyset)
\]

This predicate holds whenever there are no keys in common between (1) the old and new create sets of \(S_a\) and (2) the use set of \(S_b\). This statement can be divided in two parts:

• In the previous compilation run, there was no information exposed by \(S_a\) that influenced any consistency checks on \(S_b\). Therefore, the definition \(S_b\) was independent of the previous version of \(S_a\), concerning the typechecker.

• During the current compilation run, there was no information exposed by \(S_a\) that influenced any consistency checks on \(S_b\). Therefore, the definition \(S_b\) is independent of the modified \(S_a\), concerning the typechecker.

As described in definition 3, incrementally typechecking a single definition takes the source definition as input, as well as the create sets for all other definitions. But during typechecking, not all information contained in these create sets is actually used. The information that is not used could just as well be omitted from the input.

**Definition 7.** The set of information used for typechecking a definition can be described as a subset of the create sets of all other definition:

\[
\text{SIG}_\text{IN}_i = \{(k, v) \mid ((k, v) \in \text{CREATE}'_j \mid j \neq i) \land k \in \text{USE}_i\}
\]

The set \(\text{SIG}_\text{IN}\) is a subset of the union of all create sets for other definitions. It describes exactly these entries of the typechecker input that are actually used in order to determine semantic validity of the typechecked definition. The \(\text{SIG}_\text{IN}\) set can be calculated only after performing constraint checks on a definition, as its formula uses the use set of the definition.

The predicate 6 holds when no information created by \(S_a\) was or is used by \(S_b\); in other words, the \(\text{SIG}_\text{IN}\) set of \(S_b\) does not contain any information related to \(S_a\). Therefore,
3. ALGORITHMS FOR SELECTIVE RECOMPILED

after typechecking $S_a$, the inputs $S_b$ and $SIGIN_b$ have not changed. There is no need to perform constraint checking on $S_b$ during this recompilation.

This method operates on the "keys" of the context information store. While correct, this granularity has shown to be relatively coarse. Consider the context information store for the key CLASS, which holds a list of names of classes in the input program. Each input file that declares a class will add a value for this information key. Each input file that uses a class or performs checks on class names reads the list of values. Using this method to detect inter-definition dependencies, all of such definitions are considered dependent on each other.

Even worse, this method triggers dependencies by the mere sharing of data. It does not consider the values that are written to the information store. When the typechecking of a definition results in the addition of a value for a specific key, this might result in other definitions being selected for typechecking. However, when this value is equal to the value that the same definition added during the last compilation run, there is no actual change in signatures. Method 2 shows a more sophisticated approach to detect dependent definitions.

**Method 2. Value-based Granularity**

After recompiling a single definition $S_a$, we can compare $CREATE'_a$ and $CREATE_a$ to see which parts of the signature has changed. If any of these information keys was lookup up by another definition $S_i$, (in other words, if any of these keys is contained in $USE_i$), we need to recompile $S_i$.

**Definition 8.** Define $DIFF_a$ as the set of information key/value pairs for which a declaration of $S_a$ has set a value that is different than the value that was set in the previous compilation$^4$:

$$DIFF_a = (CREATE_a \triangle CREATE'_a)$$

**Definition 9.** Consistency checking of an unmodified definition $S_i$ is only necessary if this predicate holds for another definition $S_j$:

$$\exists k, v(k \in DEP'_{j \rightarrow i} \land (k, v) \in DIFF_i)$$

With this algorithm, recompilation starts with the set of modified definitions and cascades to unmodified definitions that need to be rechecked. Cascading is triggered when a value in the store has changed with respect to its value during a previous compilation of that definition.

The following scheme concretizes the incremental typechecking functions $TC$ using the value-based granularity predicate for selecting dependent definitions. The input program consists of input files $S_{i,j,...}$. Let $MOD$ specify the set of files which have been modified since last compilation: $MOD \subseteq S$. Likewise, let $UMOD$ define the set of unmodified definitions since the last compilation run. Therefore, $UMOD \subseteq S$, $MOD \cap UMOD = \emptyset$ and $MOD + UMOD = S$.

$^4\triangle$ denotes the symmetric set difference
3.6 Transformation Algorithms

1. For each file in $\text{UMOD}$, read the cached $\text{CREATE}$ set.
2. For each file in $\text{UMOD}$, read the cached $\text{USE}$ set.
3. For each file in $\text{MOD}$, read the file and parse it, creating a set of input ASTs $S_{i,j,...}$.
4. Perform the function $\text{TC}$ with inputs $S_{i,j,...}$, $\text{CREATE}$ and $\text{USE}$ use. This results in $\text{ERR}$, the set of errors for the input files $S_{i,j,...}$, and a pair of sets ($\text{CREATE}$, $\text{USE}$) for each input file.
5. Write the newly generated $\text{CREATE}$ and $\text{USE}$ sets to the cache (disk).
6. Using the predicate from definition 9, determine if any files from $\text{UMOD}$ have to be reconsidered for consistency checking.
7. Let $\text{DEP}$ be the set of files from $\text{UMOD}$ for which the predicate holds. Assign new values to $\text{MOD}$ and $\text{UMOD}$: $\text{MOD} := \text{DEP}$ and $\text{UMOD} := S \setminus \text{DEP}$. In other words, consider the files in $\text{DEP}$ as modified and all other input files as unmodified.
8. If $\text{DEP}$ was empty, continue to the next step. Otherwise, continue from step 1 using the new $\text{MOD}$ and $\text{UMOD}$ sets.
9. Let $\text{ERR}'$ be the union of all $\text{ERR}$ sets gathered in step 4. This is the result of the incremental typechecker: a list of violations of the compiler’s consistency checks.

### 3.6 Transformation Algorithms

After the input program is validated by the typechecking stage, our compilation model prescribes a model-to-model transformation stage [26]. Each transformation is a replacement of a node in the abstract syntax tree. This stage incrementally transforms the source AST to an AST with nodes of the target language of the compiler. For example, a Java-to-bytecode compiler would transform an integer addition in Java to a list of bytecode instructions.

Figure 3.12 shows an abstract syntax tree in the three consecutive versions during the model-to-model transformation stage. The first AST represents an example of a Java AST, which describes an expression containing the addition of two integer values. The second AST is partly rewritten (transformed): only the integer nodes are replaced with their corresponding bytecode variants, exemplified by $\text{LOAD}$ instruction nodes. In the last AST, the expression is transformed into a sequence of AST nodes describing bytecode instructions, which can be converted to binary bytecode instructions and written to disk as output of the compiler.

Transformations can be described using rewrite rules. Such a rule consists of an input node to match, an output node to use as replacement and a number of conditions. These three properties of a rewrite rule can be described in more details as follows:

- A pattern to match as input node. Such a pattern may match on the type of input node, for example “each node that represents a $\text{Plus}$”, or entail a more complex predicate on the input nodes.
• The output to generate. This may either be a fixed node, like `Plus(3)`, but in many cases is described as a template in which parts of the input node may be used. For example, the output of a rewrite rule can be `Plus(x, y)`, in which `x` and `y` are dependent in the input node.

• A list of constraints that furthermore restrict the application of the rewrite rule. Context information can be used to dictate whether the rule is applicable. For example, a rewrite rule could match any function call, but only be applied when the called function returns a specific type.

The syntactic format of a rewrite rule and its conditions depends on the implementation language of the compiler. Procedural or object-oriented languages do not provide rewrite rules or node matching as primitives of the language. In such languages, transformations have to be described explicitly, given a rule and a node:
3.6. Transformation Algorithms

- Perform the necessary matching to decide whether the node matches the rule’s input pattern.
- Check whether all conditions present in the rewrite rule are fulfilled.
- Construct the output node following the rule’s output template, and replace the AST node with the new node.

Transformation languages are programming languages specifically designed to perform these rewrites rules or transformations. For example, Stratego/XT, described in more detail in chapter 4, provides language primitives to perform pattern matching and replacement of AST nodes.

Rewrite rules can be classified in a number of ways [52]. For our purposes we will distinguish the following transformation types:

1. **Local source to local target.** Part of a definition is replaced with different content. Constant folding and constant propagations are simple compiler optimizations that can be implemented using local-to-local rewrite rules, as long as such optimizations are restricted to application within a single definition.

2. **Global source to local target.** In this type of transformation, multiple source definitions translate to a single program element of the target language. An example is function inlining. The input to the rewrite rule is both a function declaration and a call site. The rewrite rule replaces the call site with the contents of the function declaration. When the call site and the declaration site are contained in different definitions, function inlining is a global-to-local transformation.

3. **Local source to global target.** If a single source node translates to a number of nodes in different definitions in the target language, the transformation is called local-to-global. Static weaving of aspect-oriented languages is a notable example; the advice is defined once in the source language but translated to any number of advice invocations.

4. **Global source to global target.** When the input of a transformation is contained in multiple definitions, and the output is spread across definitions as well, the transformation is classified as global-to-global. As an example, many optimizing compilers analyze the complete source program and transform the output based on heuristics. When function inlining is performed automatically by the compiler, based on function size, number of call sites and other metrics, the compiler is applying a global-to-global transformation.

An interesting observation for selective recompilation is that local-to-local transformations do not impose any dependencies. These transformations can safely be cached because their scope is limited to one definition. Each of the other transformation types impose dependencies which have to be tracked in order to guarantee correct selective recompilation.
3.6.1 Innermost Transformations and Compilation Order

The dependency tracking algorithm for typechecking (described in the previous section) describes two phases: (1) an analysis phase in which global information is gathered, and (2) a constraint checking phase in which this information is used to validate parts of the syntax tree. This style of typechecking is a two-pass traversal, because each element of the syntax tree is traversed twice. The first traversal will record signatures and remember all declaration sites of global functions, variables, classes and other constructs. A second traversal is able to rely on the presence of all global information in the context information store. For example, during the first pass all attributes of a class have been recorded, including those defined in inter-type declarations of aspects. Using the static weaving approach, the typechecker matches aspect code to the definitions on which it applies (join points) and then “pulls in” any inter-type declarations from the advice.

Looking at transformations of the source tree, some transformations can be implemented using a single-pass traversal of the source program. For example, function call qualification, partial evaluation and lifting of list comprehensions to functions can be done during a single traversal through the program. However, there are cases in which the features and semantics of a language require a more sophisticated ordering of transformations on the program.

Most rewrite rules will reduce part of the AST to a “simpler” version, containing constructs that relate closer to the target language. The rewriting ends with an AST that contains only target language constructs. A single AST node can be rewritten any number of times before resulting in the target language. This allows rewrite rules to be simple and concise: a single rule performs a single transformation, oblivious of the complexity of the transformation system as a whole.

Not all rewrite rules are reducing rules. Rewrite rules can also be staged or ordered. We’ve previously described function inlining as a local-to-global transformation. In a particular implementation, one rewrite rule matches “small” function definitions, that is, functions containing only a few constructs. This rewrite rule removes the definition from the AST but remembers (in the information store) the function definition and the fact that this function must be inlined. A second rewrite rule matches a function call and replaces this call with the function body, if the function was designated to be inlined. This two rewrite rules can be described in pseudo-code:

1. **Match:** Function(name, statements*).
   **Condition:** count(statements*) < 10.
   **Output:** remove the node, write to the information store: "IsInlined" → name and ("FunctionDecl", name) → statements*.

2. **Match:** FunctionCall(name).
   **Condition:** the information store contains a value name for the key IsInlined.
   **Output:** statements*.

This rewrite rules perform function inlining in two stages. After replacing the function call with the function body, other rewrite rule can match the statements of the function body.
In fact, each transformation during the model-to-model transformation phase has the potential of triggering any number of new transformations. The set of transformations in a compiler however does need to be exhaustive; else, infinite transformation occurs.

The exact ordering of nodes to consider during the transformation stage is not relevant for our algorithms, as long as the ordering is consistent over multiple compiler invocations.

The compiler does need a termination criterion for the transformation phase. When one full pass over the program’s AST tree does not provoke any applicable transformations, the compiler can be sure that transformation has finished; no further iterations can possibly cause any change and compilation is finished.

For incremental recompilation, we impose another restriction on the transformations applied by the compiler: the transformations need to be deterministic. That is, given a node in the source AST and a state of the context information store, the result of a transformation must be fixed. There can be no use of random- or time-based information in a transformation rule.

Summarizing, our algorithms assume the following restrictions on the set of transformation rules of the compiler:

**Definition 10.** The set of transformation rules of a compiler must fulfill the following properties:

1. The transformations are exhaustive or congruent. Repeated application of the rewrite rules must terminate; at some point during transformation, no rewrite rules are applicable.
2. The transformations are deterministic.

### 3.6.2 Ordering

One particular consequence of having more than two iterations of transformation is that the transformations can be dependent not only on the source tree but also on other transformations. This introduces an ordering in the application of transformations. Consider the following scenario: three definitions $S_a$, $S_b$ and $S_c$ exist; transformation of $S_a$ causes a transformation of $S_b$ to trigger, which in turn is the cause of some transformation on $S_c$. Figure 3.13 shows the dependency graph for this situation. Using the two-phase dependency tracking algorithm, we would state that the dependencies are $S_a \rightarrow S_b$ and $S_b \rightarrow S_c$.

![Figure 3.13: Dependency graph at definition level](image)

Now consider the case in which both $S_a$ and $S_c$ have been changed and need to be recompiled, while $S_b$ is unchanged. Recompiling $S_a$ first may trigger the recompilation of $S_b$, if any value for the keys in $\text{DEP}_{a \rightarrow b}$ have changed since the last compilation. However,
in case $S_c$ is recompiled first, compilation will use the cached values $\text{CREATE}_b'$, because $S_b$ has not changed. These values recorded during the previous compilation, however, are dependent on $S_a$ which is to be recompiled. In the event of $S_a$ generating a different set $\text{CREATE}_a$, the values of $\text{CREATE}_b$ can change as well.

The problem originates in the fact that the algorithm only tracks direct dependencies, while in multi-stage transformations there can be transitive dependencies. Actually, the addition of the indirect dependency $S_a \rightarrow S_c$ in the dependency graph makes explicit the order in which the definitions have to be compiled.

Tracking transitive dependencies requires a notably more sophisticated approach of information store tracing. In essence, the algorithm has to be able to determine the validity of cached entries of the context information store. A cached entry is called valid if its value can safely be used by a transformation; that is, the creator $S_i$ of the information store entry is surely not to be recompiled. The algorithm has to be sure that transitive dependencies never cascade to $S_i$.

### 3.6.3 Causality of Rules

Transformations are able to influence one another in two ways.

First, the output $S'_a$ of a transformation $A_1$ on definition $S_a$ might be the input of another transformation $A_2$. While there is a causal relationship between both transformations, there is only one source definition involved. Therefore, this type of causality imposes no new inter-declaration dependencies.

Secondly, a transformation $A$ of a definition $S_a$ might modify the context information store in a way that provokes another transformation $B$ on $S_b$. This introduces a causal relationship between $S_a$ and $S_b$: the compilation of $S_a$ affects the compilation of $S_b$. In other words, the application of transformation $B$ is causally related to the application of $A$.

Figure 3.14 shows three transformations $A$ on $S_a$, $B$ on $S_b$ and $C$ on $S_c$. The arrows represent creations and usages of information in the context information store. It is clear that transformation $B$ depends on transformation $A$. However, the figure also makes explicit that transformation $A$ uses information with key $K_{1,2}$ before adding a value for $K_3$. If the entries for $K_1$ or $K_2$ have a different value (or no value at all) in a subsequent compilation run, the transformation $A$ is not guaranteed to have the same output; possibly, the value for key $K_3$ is different as well. Consequently, we can state that the value for $K_3$ is dependent on the value for $K_1$ and $K_2$. Furthermore, this cascades to the fact that the value of $K_4$ is dependent not only on the value of $K_3$, but indirectly on $K_1$ and $K_2$ as well.

**Definition 11.** Let $(K_i, V_i)$ be an entry in the context information store created by definition $S_i$. $\text{IN}(K_i, V_i)$ defines the set of (key, value) pairs which have been read during the transformation of $S_i$ before the addition of $(K_i, V_i)$.

Using definition 11, a dependency graph like figure 3.15 can be created in which each node represents an entry in the context information store and incoming arrows represent elements of the set $\text{IN}(K_i, V_i)$. Because an arrow defines a “created-before” relation, the graph forms a directed acyclic graph, or more specifically, a polytree.
3.6. Transformation Algorithms

Figure 3.14: Context information creation and usage during three transformations

![Dependency Graph]

Figure 3.15: Dependency graph with information store keys

3.6.4 Cache validity

The rule tracking algorithm of section 3.5.4 needs to keep track of the sets USE\(_i\) and CREATE\(_i\) for each definition \(S_i\). For a multi-stage transformation system, we introduce another algorithm for selecting and ordering definitions to recompile. This algorithm additionally keeps track of a total ordering of all usages and creations of context store values during compilation. Table 3.16 illustrates the data that is recorded for dependency analysis.

As with the typechecker dependency algorithm, definition 9 specifies the condition for selecting unmodified definitions for recompilation. What is different in this algorithm handling transitive dependencies, is that cached values in CREATE sets are not always valid, exemplified in section 3.6.2. An entry in the create set of a definition is called valid if its cached value can safely be used for recompilation, i.e. the compiler is certain that the creation of the cached value is unrelated to or “untouched” by a modification in the source.
Figure 3.16: List of information transactions

<table>
<thead>
<tr>
<th>No.</th>
<th>Type</th>
<th>Key</th>
<th>Values</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>USE</td>
<td>(K_1)</td>
<td>([V_1])</td>
<td>(S_a)</td>
</tr>
<tr>
<td>2</td>
<td>USE</td>
<td>(K_2)</td>
<td>([V_2])</td>
<td>(S_a)</td>
</tr>
<tr>
<td>3</td>
<td>CREATE</td>
<td>(K_3)</td>
<td>(V_3)</td>
<td>(S_a)</td>
</tr>
<tr>
<td>4</td>
<td>USE</td>
<td>(K_3)</td>
<td>([V_3])</td>
<td>(S_b)</td>
</tr>
<tr>
<td>6</td>
<td>CREATE</td>
<td>(K_4)</td>
<td>(V_4)</td>
<td>(S_b)</td>
</tr>
<tr>
<td>7</td>
<td>USE</td>
<td>(K_4)</td>
<td>([V_4])</td>
<td>(S_c)</td>
</tr>
</tbody>
</table>

Definition 12. A cached entry in the context information store \((K_i, V_i)\) created by definition \(S_i\) is valid if and only if:

1. \(S_i\) is not selected for recompilation, and
2. all entries in \(\text{IN}(K_i, V_i)\) are valid.

This provides us with a recursive formula that can attain validity of a cached information entry. The first condition prevents cached values to be used for definitions that are out-of-date, that is, which will be recompiled. The second condition prevents using a value that has dependencies that are still eligible for recompilation.

3.6.5 Algorithm for selective compilation

During an initial full compilation of the program, the compiler traces the transactions in the context information store, resulting in a directed acyclic graph describing all sets \(\text{IN}(K_i, V_i)\): the context information dependency graph \((\text{CIDG})\). Each node in this graph represents either a creation or a usage, as specified in the rows of table 3.16. The following is an informal description of the algorithm for selective recompilation:

1. For each definition that has modified since the last compilation, select the definition for recompilation.
2. For all leaf nodes in the CIDG, see if they represent a creation by a definition that is currently not selected for recompilation. If so, mark the created entry as valid and remove the node from the dependency graph. Repeat until no changes to the CIDG are made.
3. Perform all eligible model-to-model transformations on definitions that are selected for recompilation. When a read request is made to the context information store, return not only the current values in the store for that key but also the valid cached values.
4. Use definition 9 to select additional definitions that need to be recompiled.
3.6. Transformation Algorithms

5. If no transformations were applied in step 3 and no additional definitions have been selected in step 4, compilation is finished.

6. For each key-value pair that was added to the context information store in step 3, check whether the newly added value is equal to the cached value. If the cached and new value are equal, remove the node corresponding to this creation from the CIDG.

7. Perform another transformation pass by going back to step 2.

3.6.6 Example Run

Continuing the example of three definitions $S_a$, $S_b$ and $S_c$ and three transformations $A$, $B$ and $C$, we’ll walk through the algorithm above.

We’ve seen in section 3.6.2 that a two-pass dependency tracking algorithm does not correctly identify transitive dependencies.

In the initial situation, definitions $S_a$ and $S_c$ are modified, while $S_b$ is unmodified compared to the last compilation run. The compiler reads the cached rule table, as depicted in table 3.16.

The first step of the dependency algorithm selects $S_a$ and $S_c$ for recompilation, because their source files have been modified. Next, looking at the CIDG in figure 3.15, the algorithm considers the leaf nodes $K_1$ and $K_2$. For this example, assume that these have been created by a definition that is not selected for recompilation. The second step of the algorithm therefore removes these nodes from the CIDG, leaving the CIDG with only two nodes $K_3$ and $K_4$. The node $K_3$ is now a leaf node, but can not be removed from the CIDG, as $K_3$ was created during a transformation on $S_a$, which is selected for recompilation.

The algorithm proceeds with step 3: perform all eligible transformations. The compiler will try to apply transformation $C$ on $S_c$, which needs the value of $K_4$. However, this value is not considered valid, as we’ve just attained that $K_3 \in IN(K_4, V_4)$ is not valid. The transformation $A$ on $S_a$ however will succeed, because the values for $K_1$ and $K_2$ are valid.

Step 6 of the algorithm will now compare (1) the value for $K_3$ that the current transformation $A$ produced, and (2) the cached value for $K_3$. If these values are the same, $K_3$ becomes valid and is removed from the CIDG, making $K_4$ a leaf node, which is subsequently removed from the CIDG during the next iteration. This allows transformation $C$ to proceed.

If the compared values however did not match, step 4 of the algorithm will select $S_b$ for recompilation. The transformation $B$ produces a new value for $K_4$, superseding the cached value. Having a value for $K_4$, the transformation $C$ will proceed.

Most importantly, we see that the transformation $S_c$ on $C$ could not be performed before the transformation $A$ on $S_a$; the algorithm induced an ordering of transformations that is required for correct operation of the compiler.
Chapter 4

The NWL language

4.1 Introduction

NWL is a small experimentation language created by the authors of Spoofax [29]. It serves a mostly pedagogical purpose of demonstrating the compiler- and editor-generation features of the Spoofax language bench. The language finds its origin in the WebDSL language [57], originating from the same research group.

The NWL language has been kept deliberately small, but feature-rich in the sense that it provides challenges for compilers and editor services. For example, the language provides access control which is provided as an aspect-oriented feature. The language contains basic semantic consistency checks and a multi-pass model-to-model transformation phase, enabling static aspect weaving. Because of the aspect-oriented nature and the static aspect weaving, the language serves as a good example of a language for which selective recompilation is intrinsically hard to support.

Globally, the NWL language provides the following components and features:

1. Data model (with entities, properties and functions)
2. Modules and imports
3. UI templates
4. Access control

These basic constructs are used in an aspect-oriented fashion; for example, access control for templates is a cross-cutting aspect that can be defined in a different source module and is weaved in statically (at compile-time).

An input program in the generalized compilation model of section 3.1 consists of a number of source files, which we call modules. In NWL, there is one entry point for the application, the main file. This main file includes other files, which may transitively include other files.
4.2 The Language

NWL is basically a simplification and re-implementation of WebDSL. Much of the syntax is re-used with little or no change. This section will show some example NWL definitions.

Each module file starts with its module declaration; the keyword `module` followed by the module name. This name must comply with the filename so that the compiler knows where to find imported modules. After the module name, imports may be specified. These are names of other modules, which are subsequently loaded and parsed by the compiler. Following the module header (name and imports), each input file consists of a number of top-level definitions; these may be data model declarations, access control declarations, UI template declarations or any other language construct that is allowed as top-level definition.

A simple one-file NWL program is shown in figure 4.1.

```nwl
module sample

// no imports

// ui declarations
define page root(a : String) {
    "a"
}

define editfoo(f : Foo) {
    action save() {
        return root("Thanks");
    }
    form {
        input(f.x)
        submit save() {
            "Save"
        }
    }
}

// data model
define Foo {
    x : String
}
```

Figure 4.1: Simple NWL program source

This sample program uses two main sub-languages of NWL: UI definitions and data model definitions. The UI definitions are specified as pages or template; a page contains
any number of page elements, and possibly includes template calls. A template again can include page elements or other templates. This allows for reusable UI components.

The data model is defined in terms of entities; an entity is a data type, which can be stored in a variable or to persistent storage. An entity declaration often relates to a database table. Each property of the entity is stored as one column of the database storage. Entities can be extended by using inheritance; a sub-entity contains all the properties of its parent entity, in addition to its own properties.

### 4.2.1 Inter-type declarations

As well as these declarative constructs, the language also provides aspect-oriented features. For example, “extend entity” constructions are allowed. These are inter-type declaration as found in aspect-oriented languages like AspectJ. It allows an entity to span multiple sources files; that is, when the entity is defined in one module, it is considered an open class, because other modules are allowed to add properties using extend entity constructs. An example of this mechanism is shown in figure 4.2.

```
module base
  entity Person {
    name : String
  }

module logging
  extend entity Person {
    last_access : Date
  }
```

Figure 4.2: Extend entity in NWL

### 4.2.2 Access control

Another aspect-oriented feature of NWL is access control. Access control is declaratively described as a set of rules that either allow or disallow access to an application’s template. Such a rule consists of any boolean expression. For real-life applications, access control rules must have access to some context information, for example: is the viewer logged in, what is the role of this user, and so forth. NWL however, being only a demo language, only supports the usage of global variables and functions in the boolean expressions. This approach is a derivative of the access control language of WebDSL [23]. In NWL, a set of access control rules is declared as follows:

```
access control rules

rule template edit_*(s : String) {
  is_logged_in() && is_admin()
}
```
The above code fragment starts with a “section identifier”, indicating the next part of the source file contains access control definitions. The section contains one rule, applicable to any template which name starts with edit_. It applies only to templates that accept a string value as first parameter, however. This selection of applicable templates for an access control rule is a typical example of a pointcut, in terms of aspect-oriented programming.

The rule specifies that only users that are logged in with administrative permissions are allowed to view these templates, given the two functions is_logged_in and is_admin are defined elsewhere.

The matching support of access control rules introduces flexibility for the developer; any template that is added later to the application will adhere to these rules, given that the naming for editing templates follows the edit_ convention. This is a feature that is typical for aspect-oriented languages. The access control rules specified in one source file are applied on all modules of the application; even if the developer of a module did not have these rules in mind (obliviousness).

4.3 Implementation

NWL is implemented using Stratego/XT [56, 9], a transformation language using simple rewrite rules. Stratego/XT possesses a combination of features making it suitable for implementing a compiler.

Given the syntax definition of the language to be compiled in SDF format [24], Stratego/XT uses a scannerless generalized LR parser (SGLR parser) to create an abstract syntax tree (AST) for the input program. The AST consists of nodes representing language constructs, variable names and literals.

A compiler is conceptually a translator between a source language (in which the developer writes code) and a target language; the target language can be low-level (machine instruction codes, Java bytecode) or another programming language. Using Stratego/XT, a compiler consists of a number of translation rules; that is, rules that match nodes in the AST and replace them with different nodes. Typically, a node gets replaced a number of times before all nodes represent language constructs from the target language.

4.3.1 Parsing a source program

Stratego provides a parser implementation that produces the AST of a textual source file, given the NWL language’s grammar definition. Syntactic errors are detected; an optional recovery mechanism is available to continue parsing even if the input file does not comply to the syntax [16].

An example AST of a module is shown in figure 4.3. The top node corresponds to the module file. Below that node, both the module header and the top-level definitions are present. Note that the exact hierarchical placement of nodes is determined by the syntax definition. This file corresponds to the source example given in figure 4.1, although the graph hides some deeper AST nodes for readability.
4.3. Implementation

Figure 4.3: The AST of a sample NWL module

4.3.2 Rewrite rules

Transformations in Stratego/XT operate on AST elements. They are easy to read and consist of a match part, a replace part and optionally a conditional. Consider the following rule, which rewrites an integer addition to a sequence of simple machine instructions:

\[
\text{rewrite_plus:}
\]

\[
\text{Plus}(a, b) \rightarrow [\text{Load}(a), \text{Load}(b), \text{Add}]\]

\[\text{where} \ <\text{is-integer}> \ (a, b)\]

The \text{Plus}(...) constructor matches a plus operation in the source language definition. \text{a} and \text{b} are free variables; they are bound to a value by Stratego/XT when the \text{Plus} node is encountered. The \text{Load} and \text{Add} constructors are elements of the target language definition. The conditional consists of a call to the \text{is-integer} rule in Stratego, which must be defined elsewhere in the compiler.

Stratego/XT also provides a more concise way of writing transformation rules using concrete syntax [55]. Transformations look more natural when they can use the source language’s concrete syntax: the syntax one would use to write the actual input program.
4. The NWL Language

The same goes for the output language, if applicable. For example, the previous example could be written as:

\[
\begin{align*}
\text{rewrite\_plus:} \\
& \text{\mid [ a + b ]} \rightarrow \text{\mid [ Load \{a\}; Load \{b\}; Add \]}
\end{align*}
\]

\textbf{where} <is-integer> (a,b)

In this example, the \textbf{[ [} and \textbf{]} operators are used to enter and exit concrete syntax mode.

4.3.3 Dynamic Rewrite Rules

Dynamic rewrite rules \cite{10} provide a way to store information outside of the AST. Dynamic rewrite rules (abbreviated to “dynamic rules”) are rewrite rules that are defined during compilation; the dynamic rules result from transformations defined in the compiler’s source code. The following example shows the addition of a dynamic rewrite rule during a transformation in the declare phase of a compiler:

\[
\begin{align*}
\text{declare:} \\
& f@\text{\mid [ function name(param\*) : ret\_type\{ body \} ]} \rightarrow f \\
\text{\with} \\
& \text{\textbf{rules}(GlobalFunc : (name, param\*) \rightarrow ret\_type)}
\end{align*}
\]

The rewrite rule matches a global function declaration in the abstract syntax tree and binds it to the variable \( f \). Furthermore, the left hand side of the rule binds the variables \( \text{name, param\*, ret\_type, body} \). The transformation does not change the syntax tree, as the right hand side (\( f \)) is equal to the left hand side; however, a dynamic rule is created that stores information about the global function. This information can be used later in the compiler pipeline to perform typechecking, for example.

The dynamic rewrite rule \textbf{GlobalFunc} is no different from a normal rewrite rule in usage. First, the rewrite rule performs a match on its input node; if this match fails, the rewrite rule cannot be applied to the given input. If the match succeeds the input term is rewritten to another term.

This dynamic rewrite rule can subsequently be used by the typechecker to determine the type of a function call expression:

\[
\begin{align*}
\text{type\_of:} \\
& \text{\mid [ x\_name(x\_param\*) ]} \rightarrow \text{rtype} \\
\text{\textbf{where} type\_* := <map(type\_of)> x\_param\*} \\
& \text{\quad \text{; rtype := <GlobalFunc> (x\_name, type\_*)}}
\end{align*}
\]
4.3. Implementation

When no dynamic rule has been created for the given function name and its types, the call to `GlobalFunc` will simply fail and the condition of the type-of rule becomes untrue. In this case, the fact that the type-of rule does not produce a type determines a typecheck error; the function call references a function that does not exist.\(^1\)

Dynamic rewrite rules follow the same scoping as normal rewrite rules: they are global to the Stratego program. In many cases, rewrite rules are storing information that is only locally relevant. This is where scoped dynamic rewrite rules have their application.

Scoped dynamic rules [10] have limited lifetime, as specified by a scoping operator. In its simplest form, they can be used very well like local variables in a procedural programming language.

4.3.4 Typechecking

After the input program has been successfully parsed, the compiler performs a typecheck phase. This phase consists of two steps, as described in section 3.5: declaration and constraint checking. When no (fatal) syntactic or semantics errors have been found in the input, the compiler proceeds with the model-to-model transformations. These are declarative rules which incrementally transform the input program to the output language. The output language of NWL is a Java-like demo language, which serves no practical goal besides being human readable.

The declaration step must collect all relevant global and context-sensitive information about the input program. For example, which functions are defined, what type of arguments do they accept, which data types exist. The AST of the input program is traversed in a top-down fashion, applying the `declare` rewrite rule on each node. Typically, this rewrite rule does not modify the AST; it is used for its side-effects, namely storing information about its input node.

During the traversal of the input program, information gathered must be stored for the constraint check phase to be used. In Stratego, dynamic rewrite rules are designed to provide the storage mechanism for this purpose. The following rewrite rule “declares” a function by storing its type information in a dynamic rewrite rule:

```plaintext
declare:
  f@Function(name, args*, body) -> f
  with
    types* := <map(arg-to-type)> args*;
  rules(
    FunctionArguments : name -> types*)
)
```

The `arg-to-type` rule transforms a function parameter AST node to its formal type (strips the argument name, which is not relevant for typechecking). The rule then creates a dynamic rewrite rule named `FunctionArguments` that maps the function name to

\(^1\)Note that this simplified rule does not take polymorphism in account.
4. THE NWL LANGUAGE

its formal argument types. Note that this dynamic rules maps closely to our definition of an information store: it provides a map of key-value pairs that is stored globally for a compilation run.

After the collect phase, all of the functions have been recorded in the information store as dynamic rules. The consistency check phase uses these dynamic rules to perform assertions on function calls:

```plaintext
check:
  FunctionCall(name, args*) ->
  "No function exists with this name"
  where not(<FunctionArguments> name)

check:
  FunctionCall(name, args*) ->
  "Incompatible argument types"
  where
  formal* := <FunctionArguments> name;
  actual* := <map(type-of)> args*;
  not(<eq> (formal*, actual*)
```

The first rule simply asserts that a function call refers to a function name that has been declared. This check is performed using the `FunctionArguments` dynamic rule; if there is no such rule defined for a given name, there is no function declared.

The second rule compare argument types. For simplicity of the example, the sample language does not support overloading or subclassing. The formal arguments of a function must exactly match the types of the given parameters. The `type-of` rule uses type information gathered from the collect phase to identify a parameter's type; the parameter could be a literal, a variable or any other expression.

4.3.5 Model-to-model Transformations

The NWL language contains high-level language constructs like access control and data model definitions that are “flattened” in the target language (Java), because the target language does not support such constructs.

Static weaving of access control rules is performed during the model-to-model transformation phase of the compiler pipeline. For each access control rule, the compiler introduces a generated global function that performs the boolean check; this function is called the predicate. Then, for each template, the compiler checks which of the access control rules are applicable. This translates to (a number of) function calls that are executed in the target application when a template is viewed. The conjunction of the boolean value results control whether a viewer should be allowed or disallowed. Detailed spared, the Java output for a simple template in NWL with the above access control rule has the following form:
public class edit_user_Template() {

    public void view(String s) {
        if ( !is_logged_in() || !is_admin() )
            throw new AccessDeniedException();

        ...
    }
}

The actual implementation of access control weaving using a multi-phase model-to-
model transformation is complicated mainly because of the wildcard matching. The next
examples do not consider wildcard matching, as this simplifies the implementation and
allows us to focus on the inter-definition dependencies are created by the transformations,
as these dependencies are the input for the selective recompilation algorithm.

The weaving is a two-phase process. During the first application of the model-to-model
transformation rules, access control rules in the source syntax tree are matched and stored
in the information store\(^2\), hereby creating a dynamic rule for each access control rule that
contains the matching pattern for templates and an identifier for the predicate.

desugar: AccessControlRule(name, args, expr) -> Nothing()
    with
        predicate_name := <string-concat>
            ["predicate_", name];
        <emit> ||
            function {predicate_name} () : Boolean {
                return {expr};
            }
        ] ||;
    rules(
        ACRule :+ name -> predicate_name
    )

This rewrite rule from NWL matches an access control rule that is found in the input
program. First, it “emits” a predicate function, which is specified in the transformation
using concrete syntax. An emit is the generation of a new top-level definition during the
transformation of an input program; the definition will be considered by the compiler similar
to an input source definition. The definition becomes part of the input program. It will match
further transformation rules and is translated to Java code by the back end. However, it will
not be checked for consistency, as the typechecking phase has already been performed. It is

\(^2\)This can be done in the typechecking phase as well.
up to the designer of the compiler to make sure there are no semantically invalid definitions emitted.

Next, the rewrite rule creates a dynamic rule called ACRule. The operator in the rule creation block is \(+\) instead of \(\cdot\) which we described before; using the \(+\) operator, multiple values can be stored in the information store for one key.

The result of this transformation is the Nothing() construct; this effectively makes sure that this rewrite rule is performed only once for each access control rule in the source tree.

This actual weaving of predicate calls in template bodies is performed by the rule described next. Matching a template, the rule first retrieves the access control rules that match this template name\(^3\). The bagof prefix is a Stratego built-in to retrieve all values corresponding to a given key in the information store.

\[
\text{desugar: Template(name, args, body) ->}
\begin{align*}
\quad & \text{Template(name, args, newbody)} \\
\quad \text{where} \\
\quad & \quad \text{ac_rules := } \text{<bagof-ACRule> name;} \\
\quad & \quad \text{predicates := } \\
\quad & \quad \quad \text{<map(predicate-name-to-function-call)> ac_rules;} \\
\quad & \quad \text{newbody := } \text{<concat> (predicates, body)}
\end{align*}
\]

For each matching access control rule, the value retrieved from the information store is the name of the global predicate function. As the name indicates, the predicate-name-to-function-call rule takes a predicate name and rewrites to a function call in the NWL language. The predicates variable now contains a list of function calls to the predicates, which is prepended to the body of the template.

One final consideration is made in the transformation rules for access control. The desugaring rule described above does not terminate; the innermost traversal of section 3.6.1 would keep applying the desugaring rule to each template infinitely. Access control predicates need to be weaved in a template only once. The information store must therefore store whether a predicate was already applied to a template.

\(^3\text{This is where we simplify as for the actual NWL language, pattern matching must be applied for the template name and argument type matching must be performed.}\)
In chapter 3 we’ve introduced an algorithmic approach to selective recompilation using automatic dependency tracking. The algorithms are independent both of the compiler implementation language and of the language under compilation. However, the compiler must adhere to the generation compilation model described in section 3.1.

Chapter 4 described the pedagogic NWL language and its implementation. In this chapter, we’ll show that the NWL compiler closely adheres to the generic compilation model, and therefore the selective recompilation algorithms are applicable to this compiler. As an example of our technique, we will present a Stratego/XT library that implements these algorithms. It can be used for any compiler implemented in Stratego/XT that follows our generic compilation model.

5.1 Stratego/XT and the Compilation Model

Stratego/XT is a term rewrite language. The generalized compilation model described in section 3.1 models the operation of a compiler as the application of transformations to the input, resulting in a program in the output language of the compiler. This section describes how compilation using Stratego maps to the features presented in the generalized compilation model.

5.1.1 Information Store

Besides performing transformations, a compiler needs to keep track of information about the program’s semantics along the compilation pipeline. In the compiler model this storage was dubbed the “information store”; a generic key-value store that retains global information.

For a compiler implemented in Stratego/XT, global information can be stored in dynamic rules. These rules can be created, applied and removed. These three operations on dynamic rules fit nicely with the set of operations we’ve defined for the information store.
Rule creation

Creating a dynamic rule translates to adding an entry to the compiler’s information store. Figure 5.1 shows a rule addition in Stratego. This example rewrite rule is used to declare the signature of a global function called `plus`, which takes 2 integer parameters and returns an integer value.

```plaintext
rules (GlobalFunc : 
  ("plus", [ Integer(), Integer() ]) 
  -> Integer() 
)
```

Figure 5.1: Rule creation in Stratego

This dynamic rule takes a tuple of (function name, argument types) as input, and rewrites this tuple to the return type of the function. During type analysis, the rule is applied to discover the resulting type of a function call. As with normal rewrite rules, multiple rules with the same name can be defined. All rules with the same name will be applied to the input term, until one succeeds in rewriting. For this particular dynamic rule, this allows function overloading.

For each dynamic rule name and corresponding match term, there is one entry in the information store. Dynamic rules use a two-level map, with the rule name as first index and the match term as second index. The information store as defined in the generalized compilation model uses a simple key-value map. Therefore, the information store is a “flattened” view of the dynamic rule set. Table 5.1 represents the information store as a table of key/value pairs, after two global functions have been declared using the Stratego code from figure 5.1:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&quot;GlobalFunc&quot;, &quot;plus&quot;, [Integer(), Integer()])</td>
<td>Integer()</td>
</tr>
<tr>
<td>(&quot;GlobalFunc&quot;, &quot;plus&quot;, [Float(), Integer()])</td>
<td>Float()</td>
</tr>
</tbody>
</table>

Table 5.1: Context-store content after two declaration of two global functions

Rule removal

Besides adding new rules to the dynamic rule set, a compiler is able to remove existing rules. The removal (undefined) of a dynamic rule corresponds to the removal of a specific
value in the list of values for the dynamic rule entry in the information store.

**Rule application**

The application of a rule corresponds to a **READ** operation on the information store. For a rule application of a rule named \( R \) on input term \( T \), the entry \( (R, T) \) is read from the information store. Depending on the application operator used in Stratego, the application results in either the first or all values from the information store.

## 5.2 Information Store Tracibility

The incremental compilation algorithms require the tracing of reads from and writes to the information store. For the dynamic rules of Stratego, no such functionality existed in the Stratego language or Stratego runtime. We’ve added this functionality to the Stratego language in order to provide information store tracibility, a prerequisite for our dependency tracking algorithms.

### 5.2.1 Stratego/XT Implementation

The Stratego compiler toolset internally uses the ATerm library to represent terms. An AST node is represented as a tree of terms. Each term can be either a primitive (number, string) or a constructor. A constructor has a name (a string) and any number of arguments; the number of arguments is the arity of the constructor.

For parsing text files to their ATerm representation, Stratego uses a SGLR parser. This parser needs a syntax definition in the SDF language. Given a string and a syntax definition, the parser produces a tree of terms which is referred to as the AST.

A rewrite rule in Stratego can perform transformations on the AST. A simple transformation matches a term and produces another term. A rewrite rule is able to use other rewrite rules, but also Stratego **primitives**. These are basic operations on terms. For example, adding two integers, concatenating strings or reading a file are Stratego primitives.

The Stratego compiler has three versions. The first version is called **strc** and translates a Stratego program (a set of rewrite rule) to a C source file. This C source file can be compiled and linked together with runtime ATerm and SGLR libraries, creating an executable that performs the transformations described by the Stratego program.

The second version is **strj**. This compiler version outputs a set of Java files, one file for each declared rule. This compiler uses a Java version of the ATerm library, provided by the Spoofax Language Workbench [30].

The third and most recent version is a pure Java implementation, exclusively developed for use by the Spoofax Language Workbench.

Besides the Stratego compiler, the framework consists of a library providing a large number of commonly used, general applicable Stratego rules.
5. LANGUAGE INDEPENDENT LIBRARY

Rule overriding

Multiple rules with the same name can be defined in a Stratego program. Each rule specifies a left-hand side pattern to match. When a transformation is applied to a term, all rules are tried consequently until one succeeds.

```plaintext
add: Integer(x) + Integer(y) -> Integer(z)
  with z := <integer-add> (x, y)

add: Float(x) + Float(y) -> Float(z)
  with z := <float-add> (x, y)
```

In some cases, the re-definition of a rule should not be treated as an addendum, but as a replacement. When using libraries of Stratego code, the need can arise to customize certain behavior of the library’s internals. In Stratego, this is archived by replacing a rule (implemented by the library) with a new version.

Rule replacement is called rule overriding in Stratego. The override keyword denotes a rule as being the replacement for another rule with the same name.

As a simple example, consider a Stratego library which performs data operations on its input and writes the output to a file. The library does not provide any way to process the data without writing to a file. But using the override mechanism, a Stratego developer can modify the behavior of the library by overriding its rules for writing files. This code example replaces the rule that writes output to a file with a rule that prints the term on the console:

```plaintext
override write-file: term ->
  <print-stdout> term
```

Rule overriding was added as a feature to strj around April, 2009. This mechanism proved to be useful for hooking in Stratego’s dynamic rule code. However, rule overriding was never backported to the strc compiler. To be able to use our separate compilation library in both strc and strj, we implemented rule overriding in strc.

The syntax definitions for rule overriding were already present, because of the usage in the strj implementation. However, the strc compiler did not have any rewrite rules for translating overriding rules to C code. The first step was to include semantic checks for rule overrides; for example, to provide an error message when a non-existing rule is overridden.

For the generated C code, the original strc compiler created one C function for each strategy. For the new implementation, each strategy is represented by a function pointer and a function implementation. The function pointer is initialized with the address of the implementing function; however, when an override is encountered, the function pointer is overwritten by the address of the function of the overriding strategy.

---

1Source can be found on https://svn.strategoxt.org/repos/StrategoXT/strategoxt/branches/strategy-overriding.
5.2.2 Tracing of rule modifications

The library for selective recompilation implements tracing of rule modifications during a compiler phase. Basically, the library hooks into the Stratego library calls that modify the hash table of dynamic rules. These are a number of ways to add/remove dynamic rules, but most methods of access to the dynamic rule hash table end up calling through to these three rules of the Stratego Base Library:

1. `dr-add-rule`, which adds a value for the given key in the rule hash table.
2. `dr-set-rule`, which modifies an existing value for the given key in the rule hash table.
3. `dr-undefine-rule`, which removes all values for the given key in the rule hash table.

By overriding these three Stratego library rules, the library is able to monitor all modifications to the set of dynamic rules during a compiler phase. However, while these overrides ensure the processing of all reads and writes to the context store, it incurs a significant performance hit, because the library code is called for each access to a dynamic rule. Therefore, our library code has a global flag that enables or disabled rule tracking, defaulting to disabled mode. This following code exemplifies the usage of rule overriding to monitor writes to the context information store of Stratego:

```stratego
// Override the Stratego base library strategy
override dr-set-rule(|name, key, value) =
    // Call the original strategy to perform the actual write
    proceed;
    // Check whether rule tracking is required
    if is-rule-tracking-enabled then
        // Store the rule creation for dependency tracking
        <record-rule-creation> (name, key)
    end
```

The library provides methods to enable and disable rule tracking. By enabling rule tracking, the library hooks both reads and writes to the context information store. The multi-phase dependency algorithm uses this method to dynamically resolve reads to data that is not actually in the context information store, but can be read from cache (described in section 3.6).

**Automatic signatures**

By hooking the primitive Stratego rules, the library is able to monitor which rules have been added to the hash table of dynamic rules while the compiler performs transformations.
A more simple implementation is used for the typechecking phase of the compiler, in which we need to determine the signature of a definition. By surrounding the declaration of a definition with `start-record-dynamic-rules` and `end-record-dynamic-rules` respectively, the library automatically provides the signature.

Calling `start-record-dynamic-rules` will record the entire current state of Stratego’s dynamic rule hash table. During the declaration of the definition, the compiler creates dynamic rules to store global type information. The set of dynamic rules created between the start- and end library calls is the signature of the definition, i.e. the `CREATE` set of our typechecker algorithm.

The implementation of these two calls for starting and stopping dynamic rule recording in our library is based on the concept of scoped dynamic rules, as explained in section 4.3.3.

### 5.3 Library Implementation

The language-independent library for selective recompilation helps a compiler developer by providing implementations of the algorithms described in chapter 3. Also, the library provides helper routines for parsing files using a parse-cache, an implementation of the context information store and automatic detection of stale cache files.

The goal for the library is to provide as much ready-to-use functionality as possible, without making the library language-dependent. In the next sections, we will describe in more detail what functionality the library provides and how this can be incorporated in a sample language that we call NWL+Aspects, or shortly “NWL”, which was described in chapter 4. This language does not actually produce usable code but does have the aspect-oriented features that are challenging for incremental compilation.

#### 5.3.1 Parsing

A whole-program compiler needs to collect all of the input files of the program to be compiled and parse them one-by-one. When the main file is parsed, it is traversed for include constructs; a node that indicates the inclusion of a module is then replaced with the parsed AST of the included module, as shown in figure 5.2.

For an incremental compiler, there is neither the need nor the intent of storing the AST of the whole program in memory. Parsing files incurs performance costs, varying between parser techniques and implementations. Minimally, costs are linear in both space and time; the files need to be read from disk (time) and stored in memory (space).

All common operating systems and file systems keep track of the modification time of a file\(^2\). The incremental compiler creates a “stamp-file” for each input file that it parses. By comparing the modification time of the stamp-file to the modification time of the input file, the compiler can identify whether an input file is modified since it was parsed during a previous compilation.

For each input file, we now have two states:

\(^2\text{With varying precision however; mostly, a millisecond or greater precision is used, with the notable exception of Windows using a FAT file system, which stores file modification dates with a 2-second precision [43].}\)
1. Unmodified; this file is not modified since last compilation. It may not have to be recompiled. However, advice code from a different source file that is weaved in to this file during a later stage of the compiler pipeline may cause recompilation of the file nonetheless.

2. Modified; this file has changes in any of the definitions it contains. It must be re-parsed.

First, note that the stamp-file approach uses a file-based granularity for parsing. A file contains any number of top-level definitions as defined in 3.1.1. When a file a modified, the compiler presumes that all of its containing definitions have changed. With more bookkeeping, this assumption can be made more precise; after parsing, the compiler can perform a side-by-side comparison of the definitions in the AST to the AST of the previous compilation run to detect exactly which definitions have changed.

Another issue arises concerning imports. The compiler only receives the name of the main file. In case this main file has not been modified, it does not need to be parsed; but then the compiler has no input at all. If the main application file contains includes, these will have to be checked for modification. The same goes for unmodified modules; they also can contain included modules that are modified. Therefore, the stamp-file of an input file not only serves as a last-parsed timestamp, but also contains a list of included module files.
Furthermore, besides modifying source files, there is another scenario in which recompilation is necessary. If the compiler itself is modified (updated to a new version, for example), there is no guarantee that an input file produces the same output compared to compilation using the older compiler version. We also include the compiler version in the stamp file. When this stored version does not match the current compiler version, the file is marked as modified. The complete contents of a stamp file is now:

1. Its modification time (stored as file metadata by the operating system)
2. A list of included module files
3. The compiler version number

The stamp file is stored on disk using Stratego’s ATerm representation. A sample stamp file for a project containing three source files is shown in figure 5.3:

```latex
("9.7pre4555",
 [ Imports("template/main"),
   Imports("ac/main"),
   Imports("pages/main")
 ]
)
```

Figure 5.3: Sample ATerm stamp file

The load-modified-modules rule in our separation library is the entry-point for all parsing. It receives the name of the main file and performs the following steps:

1. Check for an existing stamp file corresponding to this source file. If one exists, read it and verify the contained compiler version. If there is a version mismatch, discard and remove the stamp file and continue as if there was no stamp file.
2. If a stamp file was found, see whether the modification time of the stamp file is newer than the modification time of the input file it represents.
   a) If the stamp file was newer, we have an up-to-date cache and the file is unmodified relative to the last compilation. Read the cache’s contents and store the create sets for the file’s definitions in memory. Note that the actual input file is not read and the compiler does not need to store the AST’s of the definitions. Recurse from step 1 for all imports mentioned in the stamp file.
   b) If the stamp file was older, the file has modified since last compilation. Read the stamp file to obtain CREATE’, the previous create sets, to gain the ability of comparing old and new create sets for a modified definition.

---

3We’ve formatted (pretty-printed) the ATerm for readability; the actual file does not use newlines or indentation.
3. If an up-to-date create set was not loaded for the file; that is, either there was no stamp file, or it was outdated; parse the file and store the AST’s of the definitions. These definitions will be typechecked. Recurse from step 1 for all imports encountered in the file’s AST.

The execution of this strategy loads a number of AST’s in memory for definitions that have been modified and will need to be re-checked for consistency. For all definitions, modified or unmodified, the cached create set is stored in memory. This information is sufficient to perform incremental typechecking on the modified files.

5.3.2 Typechecking

Section 3.5 describes an algorithm for incremental typechecking. The algorithm assumes that typechecking is performed in two phases; a collect phase in which all global- and context information is stored, and a check phase in which the language constraints are validated.

During the first compilation of an application, there are no caches, dependency files or stamp files. The compiler must resort to a whole-program compilation. Besides the stamp files that result from parsing, the algorithm describes the create- and use-set for each top-level definition.

The create set for a definition describes the signature of the globally visible constructs in that definition. The create set of a definition $S_a$ is required when we need to typecheck another definition $S_b$ that “uses” constructs (methods, types) from $S_a$. When $S_a$ is modified, it is recompiled and the signature information is stored during the first phase of typechecking. When $S_a$ is not modified, its stamp file will be used during parsing to check for imports. Because the compiler will not recompile $S_a$, it needs to load the create set from disk. In the library implementation, the create set is stored alongside the import and version information in the stamp file.

Using the cached create sets of unmodified definitions, the compiler is able to typecheck the modified definitions without parsing the whole program. However, the algorithm describes that typechecking can be transitively induced; a modification to one file may result in typechecking errors in another file. Therefore, after typechecking the modified files, the compiler needs to perform a dependency check. This check uses the create- and use sets of the definitions, and results in a list of files that have to be rechecked for errors.

The following pipeline describes all the steps needed for the NWL compiler to implement incremental parsing and typechecking:

1. Use `load-modified-files` to load AST’s for modified definitions and create sets for unmodified definitions in the input program.

2. Simulate the collect phase for unmodified files by copying the signature information from the create set into the context information store.

3. Having the global signatures of all unmodified files in the information store, perform normal typechecking of the modified files. The library records all accesses to the information store, generating new create- and use sets for these modified files.
4. Write the newly recorded create sets to cache files.

5. Use the cascading-typecheck algorithm 9 to check for unmodified definitions that have to be typechecked. When there are no such definitions, typechecking is finished.

6. For each of the definitions that resulted from the execution of the cascading algorithm, parse the corresponding file. The collect phase does not need to be performed, because the compiler has already loaded the cached create sets.

7. Check these definitions for consistency. Repeat from step 5.

8. Typechecking is finished when no new definitions have been detected by the cascading algorithm in step 5. The compiler has stored in memory the use sets for each of the definitions that were typechecked during the compilation. These can be written to disk and removed from memory, as this information is only required for the next compilation run.

Let us illustrate this algorithm by tracing the execution of the compiler for a small sample application. This NWL main file `main.nwl` consists of a function definition and an import to an external source module:

```nwl
import module.nwl

function double(int x) : int {
    return x * 2;
}
```

The external module file `module.nwl` contains a template definition that uses the function defined in the main file:

```nwl
import main.nwl

template mainview() {
    "Five times two is " double(5)
}
```

One of the compiler’s consistency checks ensures that a function call refers to an existing function, and that the function accepts the given parameter types. For this check to be implemented, the compiler needs to record (declare) the functions in all modules during the collect-phase, using this rule, repeated from section 4.3.4:


For selective recompilation, the important observation is that `main.nwl` declares a function and `module.nwl` uses this declaration. This introduces a dependency for the typechecker. If the source file of `main.nwl` changes and causes the signature of the function `double` to change, `module.nwl` needs to be re-checked; even if the `module.nwl` source file has not changed.

After an initial compilation of the whole program, the use and create sets have been stored to disk. Given the example rules above, these sets are as follows:

```plaintext
CREATE_main = [    
  ("FunctionArguments", "double", ["int"])  
]
CREATE_module = []

USE_main = []
USE_module = [    
  ("FunctionArguments", "double")  
]
```

Given these sets, consider what happens when the developer changes the `main.nwl` file to contain this text:

```plaintext
import module.nwl

function double(float x) : float {    
  return x * 2;
}
```

Because the function `double` now has a floating point argument type, the call in `module.nwl` should be identified as having incompatible argument types. These steps describe how the selective recompiler operates in this case, given that `module.nwl` itself has not been modified.

1. Using operation system file timestamps, the compiler identifies that `main.nwl` has to be recompiled and parses its contents; `module.nwl` is considered not-modified and only its create set is loaded from disk.
5. **Language Independent Library**

2. The create set for the module file is loaded in the information store; in our example, the create set is empty.

3. Declaration of the main file results in a `FunctionArguments` entry in the context information store with a key-value pair of `(double, [float])`.

4. The new create set for the main file is stored to disk.

5. The dependency algorithm uses the stored use set and the new create set, and intersects these two. Because both the create set of the main file and the use set of the modules file have an entry for `(FunctionArguments, double)`, the algorithm detects that the module has to be re-checked.

6. When the consistency rules are applied to the module file, the information store indicates that the parameter type for `double` is now a floating point value; the `check` rule will correctly report the incompatible function call.

### 5.3.3 Model transformations

Section 3.6 describes our algorithm for tracking dependencies during a multi-pass model-to-model transformation of the input program. Mostly, these transformations simplify the input program to a core language. This core language is generally close to the target language of the compiler, so the back end implementation is more straightforward.

A generic compiler implementation of a model-to-model transformation phase is described for the NWL language in section 4.3.5, based on the work of [26].

#### Providing Selective Recompilation

For multi-phase model-to-model transformations, we have defined a dependency-tracking algorithm in section 3.6 that dynamically loads values in the information store which are subsequently used for compiling modified definitions.

Despite of the implementational complexity of this algorithm in the library, the usage of the library in the desugar phase of the compiler is very easy. The library takes care of all dependency handling completely automatically, by intercepting access to the information store and performing dependency tracking on the fly.

The execution model of the model transformation phase of the compiler can be described as follows:

1. Load a cache file containing dependency information generated by the last compiler run.

2. Perform an innermost traversal of the AST for each top-level definition.

3. When no more desugaring rules are matched (all applicable transformations have been applied), generate a new dependency information file.
The crux of the dependency tracking lies within the dependency file and the interception of access to the context information store. The assistance of the separate compilation library during the model transformation phase is threefold:

1. Provide automatic bookkeeping of all dependency information.
2. Provide context information for program definitions that are not loaded; that is, for definitions that are unmodified compared to the last compilation run and need not to be recompiled.
3. Trigger recompilation of an unmodified definition when context information on which the previous compilation was dependent.

Considering the access control example from before, the AST nodes for `AccessControlRule` and `Template` are examples of top-level definitions. The access control rules can be specified in a different source file than the template; this is the aspect-oriented approach of separation of concerns. For the compiler, this means that there is a dependency between the two source files. The access control rule has the role of “declaration site”, as it declares the rule. The template is a “usage site”, as it uses the declaration of the rule, although, because of obliviousness, the template source file does not reference the access control rule.

This dependency between the declaration and the usage is twofold:

1. The translation of the usage site depends on changes in the context information created by the declaration site. For example, if the access control rule is removed, the compilation of the usage site (the template) should be triggered, as the call to the predicate must be removed from the output code.
2. The translation of the usage site needs context information from the declaration site, even if the declaration site has not changed. If the declaration is not selected for recompilation (the definition containing access control rules has not changed since last compilation), this context information must be retrieved from a cache.

**Implementational Details**

The algorithm discussed in section 3.6.5 describes the usage of a context information dependency graph (`CIDG`), which basically keeps track of dependencies between entries in the context information store. This CIDG is a directed acyclic graph, in which parent nodes represent an event that happened before the events represented by their child nodes. Figure 3.16 represented this CIDG as a table, in which entries are sorted in chronological order: the rule table. The table representation differs from the graph representation in that it does not contain causality; there is no relationship between the rows, except their ordering. However, the table representation is simpler to implement and provides sufficient information to enable sound and safe selective recompilation.

The table is stored as an ATerm. The ATerm grows linearly with the amount of context information store reads and writes. This number of store operations is dependent on the
complexity of the compiler and the size of the input program. In any case, the storage size of the table can increase to enormous sizes. Therefore, it is important to implement the table access efficiently.

The selective recompilation library contains an implementation of the rule table as a flat list of rows, stored on disk as an ATerm. Because Stratego, both the C and the Java version, contains extensive ATerm support, this is the easiest to implement. For the Java version, we’ve added binary ATerm support to Stratego/XT, which was already present in the C version\(^4\). However, the ATerm library does not support partial reading or streaming of ATerms; therefore, the table has to be loaded and saved in its entirety. This boils down to the regrettable fact that, independently of the compiler’s ability to perform selective recompilation, the memory usage of the compiler still is linear with respect to size of the full input program. This is still an open issue. Some work is currently done to avoid or mitigate this issue by using an efficient “index” [36]. For the selective recompilation library we’ve partitioned the table by definition; creations and usages of a single definition are stored in a single file. This slightly slackens the problem because reading and writing of ATerms does not entangle huge files. The library however needs to now all creations during the model-to-model phase, which means that yet all tables have to be loaded in memory.

Besides storing and reading the table, the library provides automatic validity checking of rules; that is, when the compiler requests the value for a key in the context information store, the library tests whether any cached values are eligible for consumption by the compiler. The library provides two implementations to test validity: forwards validity checking and backwards validity checking.

Backwards validity checking is a direct implementation of the recursive predicate 12. When a read request is made to the context information store, the table is scanned to find creations corresponding to the requested key. For each row representing a creation, the validity is attained by executing the recursive algorithm: first check if the row itself corresponds to a definition that is not to be recompiled, then recurse to the previous row. When a creation row is known to be valid, it is permanently marked as such, because a creation never looses validity once obtained.

Forwards validity checking takes a different approach to validity checking. The library performs bookkeeping using a table pointer. This pointer is a number representing a row in the rule table. All rows before that appointed row are considered valid. The rule pointer starts at the first entry of the table and descends along the rows of the table during compilation. After each traversal of the model-to-model transformations on the input definitions, the table pointer is updated. Because validity is attained from the top of the table and increased monotonically, the check for validity is no longer recursive.

In practice, the forward method proves to be simper to implement and debug, and operates better performance-wise. The size of the table and the memory usage resulting from that are however issues that need to be solved in further work. A random-accessible mechanism is needed to allow efficient access to dependency data. A database implementation such as SQLite might provide a sufficient storage back end.

\(^4\)See chapter 9 for all software contributions of this research.
5.4 Testing

Besides the theoretical correctness of the underlying algorithms, we’ve created a test suite to verify the correct operation of the selective recompilation framework. This suite uses the NWL compiler to compile various sample projects which are crafted to utilize the selective recompilation features of the compiler.

5.4.1 Unit tests for NWL

To stress the various components of the compiler that relate to selective recompilation, we use specially crafted NWL applications. These applications consist of a main application file and a number of modules. Besides normal NWL statements and definitions, some additional syntax has been added to provide instructions for the testing framework.

All of the testing instructions start with the character sequence “-- START” and end with “-- END”. The instructions identify changes in a program, that is, one source file actually represents two versions of the source file at the same time. Our test tools can create the first and the second version of the file, given the source file and a version number.

Currently, two instructions are implemented in the test tools:

- **ONLY 2**: the NWL source code between the start- and ending marks are added in the second version (deleted in the first).
- **DOUBLE**: the section of NWL source code is present only once in the first version, but duplicated in the second version.

The first toolset contains unit tests that test whether a target file is updated when it should be, and verifies that no target files are unnecessarily regenerated.

One of the tests in the suite verifies the handling of extend entity constructs (as described in section 4.2.1). In the first compiler run, there are no cached target file, and the compiler must resort to whole-program compilation. Next, the toolset updates the program as described using the test instructions: a ONLY 2 segment of text adds an extend entity definition to the source program. The NWL compiler is invoked a second time, but now the compiler comes across cached target files and metadata generated by the selective recompilation framework. The test suite uses timestamps to assert that only one target file has changed: the target file describing the entity that was extended.

Other tests include adding an access control rule, which should invalidate all templates and pages, and changing an aspect that operates on various other source definitions.

These tests are language-specific: the list of files that should be impacted by a source code change is dependent on the language features. For example, the view page of an entity must change when the entity has been updated, which is a NWL-specific concept. The next section describes another set of testing tools which is language independent.

5.4.2 Language Independent Tests

The language-dependent rules for selective recompilation identify which source-level changes result in which changes to target file. In section 3.3 we’ve asserted that it is gen-
erally very hard for language developers to specify a sound, complete and minimal set of recompilation rules. Abstracting away from the language-dependent rules for selective recompilation, there are some properties of an incremental compiler that must hold, independent of the language definition of the compiler.

Recompilation is an optimization and must not change semantics, compared to the whole-program compiler. Simplifying the compiler functions described in chapter 3, there is a function `Compile` that takes an input program and results in an output program. Similarly, there is a function `Recompile` which takes an input program and cached data, outputting a target program, and new cached data. The first compiler invocation always results in a full compilation. Using $S_1$ for a source program and $T_1$ for the target program leads us to these equations:

- $\text{Compile}(S_1) = T_1$.
- $\text{Recompile}(S_1, \emptyset) = (T_1, C_1)$.

Now when the source program changes to $S_2$, independent of the volume of these changes, we can assert that recompilation of $S_2$ using the cached information of the first compiler run $C_1$ must output the same target program $T_2$ as whole-program compilation would:

- $\text{Compile}(S_2) = T_2$.
- $\text{Recompile}(S_2, C_1) = T_2$.

Our test suite takes a source program that is optionally annotated with test suite instructions described in the previous section. The tool performs these operations, sequentially:

1. Compile the source program (original version) using a whole-program compiler. The output program is stored as $\text{FULL}_1$.
2. Compile the source program again using the whole-program compiler, after applying the changes described using testset instructions. This output is called $\text{FULL}_2$.
3. Compile the first version of the source program, now using the selective recompiler, but without the presence of any cache files. The output is stored under $\text{SEP}_{1a}$.
4. Without changing the source files, perform selective recompilation on the source program, taking into account the cache files created by the previous run of the compiler. This output is labeled $\text{SEP}_{1b}$.
5. Using the cache files generated by the previous run, run the compiler again but now using version two of the source program, generating $\text{SEP}_2$.
6. Similarly, preserving the cache files of the previous step, revert the source code back to the first version and re-run the selective recompiler. This generates output $\text{SEP}_3$.
After these steps, we have compiler outputs $\text{SEP_{1a,1b,2,3}}$ and $\text{FULL_{1,2}}$. The following conditions must hold for these sets of output files:

1. $\text{SEP_{1a} = FULL_{1}}$, in other words, whole-program compilation produces the same output as selective recompilation.

2. $\text{SEP_{1b} = SEP_{1a}}$: recompiling an identical program using selective recompilation does not alter the target files.

3. $\text{SEP_{2} = FULL_{2}}$: sequential incremental compilation of source version 1 and 2 does not differ in output from whole-program compilation of version 2.

4. $\text{SEP_{3} = FULL_{1}}$: incremental compilation of multiple source versions provides the same output as a whole-program compilation.

If a whole-program compiler is not available, the test tools are able to simulate a whole-program compiler using an incremental compiler. Before each run, all cache files are removed from disk, so the compiler must revert to a whole-program compilation.
Chapter 6

Case study: WebDSL

The dependency algorithms described in the chapter 3 are applicable to any compiler that adheres to the requirements we have noted: (1) the compilation model conforms to the generic compilation model of section 3.1 and (2) the implementation of the compiler provides a way to track context information accesses.

WebDSL is domain specific languages with many aspect-oriented language features. The compiler is implemented in Stratego/XT, using roughly the same compilation pipeline as the NWL language described in chapter 4. WebDSL programs are translated to Java code (using Java Servlet technology) and can be deployed in any servlet container, like Apache Tomcat.

The WebDSL compiler has been implemented as a whole-program compiler, without consideration of incremental compilation or dependency tracking. Developers are observing drastic increases in compilation times for larger WebDSL programs. This makes WebDSL an ideal candidate for an empirical study of our separate compilation techniques. Figure 6.1 shows the compilation times of various WebDSL programs, plotted against the lines of code of each project (excluding comments). The compilation times for larger projects are problematic for rapid prototyping and a quick developer workcycle.

This chapter starts with a description of the WebDSL language, its history and current state. The chapter then continues to explain the implementation techniques of the WebDSL compiler; many of these techniques have been re-used for the implementation of the NWL compiler and therefore already introduced in chapter 4. A number of refactorings had to be performed for the WebDSL compiler to be compatible with the separation library. We’ll describe the mismatches between the generic compiler model and this specific compiler implementation. Section 6.4 explains how the utilization of the separation library supports the responsive integrated developer environment (IDE) for WebDSL programmers, which includes an incremental typechecker. The last section of this chapter describes the integration of the separation library into the whole-program compiler for WebDSL, which has unfortunately required an extensive amount of refactoring on the compiler.
6. CASE STUDY: WebDSL

6.1 The WebDSL language

WebDSL is a domain specific language used to develop dynamic web applications with a rich data model [57]. The language contains high-level concepts for the domain of web applications, like pages, sessions and entities. Actually, for each of the technological domains a sub-language is defined which is specifically tailored for that domain. For example, a SQL-like query-language is used for data retrieval and a declarative access control language is available. The embedding of sub-languages in a DSL is called language embedding.

Language embeddings are very common in web application development. SQL expressions and HTML fragments can be found embedded in many languages. These fragments are however not linguistically integrated [25] and therefore no consistency checking is performed by the compiler. The languages embedded in WebDSL are integrated in the compiler and statically verified during compilation.

The embedded sub-languages mostly coincide with the various aspects of a web application. WebDSL allows these aspects to be defined in different modules, like other aspect-oriented languages. A typical web application consists of a data-model module, a module containing access control rules, a user interface and styling definitions.

Besides providing a basic web development language, WebDSL contains a library of built-in web components, a templating language for user interfaces, coupling with various back end database systems (supporting indexing and searching) and an integrated development environment.
6.2 Compiler Implementation

The WebDSL compiler is implemented in Stratego/XT and transforms WebDSL files (carrying a .app extension) to a collection of Java files that conform to the Java Servlet specifications.

Globally, the compiler pipeline consists of these steps, as illustrated in figure 6.2.

1. **Parsing.** The input files are syntactically checked and parsed by means of a scannerless generalized LR parser (SGLR parser). This results in one abstract syntax tree (AST) for the whole input program.

2. **Renaming.** All identifiers (names of variables, functions and such) are renamed to unique identifiers. This prevents (variable) shadowing and simplifies the implementation of the rest of the compiler pipeline. Also, declarations are linked to usages in this step: for each unique variable identifier there is an accompanying type definition.

3. **Typechecking.** The nodes of the AST are checked against a set of language-specific constraints. If the input program does not conform to any of these constraints, an error message is raised and compilation stops.

4. **Desugaring.** Syntactic sugar was coined by P. J. Landin in [38] as a form of alternative notation for applicative expressions. More generally, syntax constructs of a language are considered syntactic sugar when they add no new semantics to the language, i.e. they do not add to the expressiveness of the language; the new constructs are merely a prettier or easier readable form of another syntax construct, making the language more “attractive” to humans. Uses of syntactic sugar ranges from simply shortening common keywords, to complex aspect-oriented concepts. In the desugaring compiler step, all syntactic sugar is translated to equivalent variants in a “base” WebDSL language, fully preserving the semantics and functionality of the program. This step of the pipeline includes static weaving of aspects.

5. **Access control.** On of the aspect-oriented features of WebDSL roots in its access control mechanism. Rules for access control can be specified separately from the actual subjects on which access control is applied. While this is implemented as a separate step in the compiler pipeline, its application is very similar to a desugaring step.

6. **Code Generation.** The steps 4 and 5 have made sure that the AST consists of only constructs from the base WebDSL language, and that all aspect-oriented features have been weaved in. The code generation step mainly consists of one-to-one translation between WebDSL constructs and Java implementation code.

The next sections describe each of the steps of the compiler pipeline. A detailed analysis is made of the impact of requiring separate compilation and using the dependency algorithms of chapter 3.
6. Case study: WebDSL

6.2.1 Parsing

A WebDSL program consists of a main application file and any number of included modules. The compiler is passed the name of this main application file and reads this file.

The SGLR parser uses a syntax definition of the WebDSL language that is defined using the Syntax Definition Formalism [54]. In spite of the interesting concepts used for composing the DSLs into one WebDSL language grammar and parser, for our discussion it is sufficient to attain that the parser provides us with an AST of the input module.

The parser operates on source files. Given the path of a source file on disk, the parser provides an AST corresponding to the source file. When syntax errors are encountered, the parser provides detailed error messages to the compiler. If the compiler, after all parsing has completed, has found one or more syntax errors, the compiler quits immediately and provides the user with a list of error messages.
6.2. Compiler Implementation

The WebDSL compiler reads the main application file and then processes imported modules transitively. This is implemented by replacing import statements in the AST with the contents of the imported modules, similar to the NWL implementation (figure 5.2). To implement selective recompilation, the parse phase is rewritten to use the library methods of processing imports. In the original compiler, the result of the parse phase is a complete AST of the input program, incorporating the AST’s of the imported modules. The rewritten compiler uses the stamp files generated by the library. For a source module that has not changed since last recompilation, only the typechecker signature is loaded; the source file is not parsed. The result of the new parse phase is a list of AST’s, one for each modified top-level definition. Also, for each unmodified definition, the stamp file is loaded, containing a CREATE set and a USE set. While the compiler previously operated on a single AST, we’ve rewritten these rules to operate on single definitions, enabling the compiler to transform single definitions independently, given the availability of signatures of external definitions.

When the parse phase completes without syntax errors, the compiler continues its front end pipeline with typechecking. This includes several substeps. During the whole typechecker phase, the library for selective recompilation keeps track of dynamic rule creations and usages to identify dependencies.

6.2.2 Renaming

After the source program has been parsed and the imports have been traversed, the compiler has a set of source definitions each defined as an AST with source language constructs. The first step of the compiler pipeline is identifier renaming. This process is described in [26] and resolves variables to their declarations. A source program can contain multiple declarations of a variable \( x \) throughout, but each usage of that variable only refers to a specific declaration. The matching of usages to definitions is specified by the scoping rules of the WebDSL language.

WebDSL replaces each declared variable name with a globally unique variable name, and updates the usages accordingly. This prevents the typechecker from worrying about variable shadowing.

For each declaration, the compiler stores a mapping between its newly created globally unique name and its type. This information is stored as TypeOf entries in the context information store.

6.2.3 Typechecking

Typechecking is implemented following the same design patterns as explained in section 3.5, using a collect phase to determine all global typing information, and a check phase to validate the source program according to the language constraints. For example, by using the TypeOf information generated by the renaming phase and stored in the compiler’s information store, each usage of a variable can easily be resolved to its declared type.

During typechecking, the selective recompilation library uses and updates the CREATE and USE sets for each definition. After typechecking the set of modified definitions (for which an AST was provided by the parse phase), the library is consulted to determine if
any unmodified definitions have to be additionally typechecked. For example, the library can deduce when a modified definition changes the signature of a function that is called from an unmodified definition. This calls for re-evaluation of the semantic correctness of the unmodified file. After the parsing phase, the library performs two operations:

- Write new signature files containing `CREATE` and `USE` sets for definitions that are typechecked.
- Identify which unmodified definitions are possibly influenced by changed signatures of the modified definitions. For these definitions, the compiler reads the AST from the stamp file and runs the typechecker on these AST’s.

The typechecker now operates in two phases. The first typechecker invocation processes only the definitions that are modified since the last compilation. The second invocation operates on the unmodified definitions that are marked by the library as dependent.

### 6.2.4 Desugaring

The desugaring phase of the compiler transforms a WebDSL input program to an functionally equivalent version using only “basic” WebDSL constructs. This simplifies the back end implementation by minimizing the set of constructs to be translated to Java code. Besides rewriting language features that can be categorized as syntactic sugar, this phase also performs static weaving of aspect code.

The compiler implementation for the desugaring phase closely follows the implementation described in section 4.3.5, where we described the NWL compiler.

Different than the NWL compiler, the whole-program WebDSL compiler performs desugaring on the AST of the whole program. Within this AST, an innermost traversal is made to desugar all nodes, as described in more detail in [37] and [9]. For the selective recompilation version of the compiler, this traversal is slightly modified. These steps outline the desugaring implementation used to enable integration with the selective recompilation library:

1. For each top-level definition that is marked `loaded` (i.e. the AST is available), perform a single innermost traversal through the AST, applying the desugar rules of the compiler to each node encountered.
2. If any application of the desugar rewrite rules succeeded, i.e. a node has been modified, repeat from step 1.
3. If the selective recompilation library has determined that unmodified (and currently unloaded) definitions have to be desugared, retrieve the AST’s from the signature files, and perform the rename phase. Repeat from step 1, including this newly loaded definitions in the desugaring process.
4. If no desugaring rule has succeeded for any node in any of the AST’s of the top-level definitions, and no new definitions have been selected for recompilation, desugaring has finished.
Similar to the typechecking phase, this is an incremental process: at the start of the desugaring phase, there is a set $A$ of top-level definitions that need to be desugared and compiled to Java code. During step 3, definitions can be added to this set, generating a larger set $A'$. At some point, there are no more model-to-model transformations eligible for the current set. This means that the program has been transformed in the WebDSL base language, and all definitions that are not in the set (unloaded) are determined to be unaffected; these definitions do not need to be recompiled, as the Java output files are present and up-to-date. For large changes to the input program, the set of definitions may grow to include all source definitions. In such a case, selective recompilation mimics whole-program compilation.

### 6.2.5 Access Control

In its current state of implementation, WebDSL performs access control weaving as a separate step in the compilation pipeline. In our generalized compiler model, we’ve identified just one transformation phase, which incorporates any number of transformation rules. The WebDSL compiler has two model-to-model transformation phases, the desugar phase and the access control weaving phase. Having multiple desugaring passes, in which different desugarings are applied, introduces implicit orderings between transformations; i.e. all desugarings are assumed to be applied before access control is weaved in. Our dependency algorithms, and therefore the selective recompilation library, do not account for such implicit ordering requirements on the model-to-model transformations. Incorporating access control in the first desugaring phase requires significant refactorings in the access control weaving code of the WebDSL compiler. As of the time writing, this is still an open issue. In section 6.5 we describe how this affects the integration of the separation library in the WebDSL compiler.

### 6.2.6 Code Generation

After the model-to-model transformations from the desugaring and access control phases, the compiler enters a new part of the pipeline: the back end. The back end of the compiler is responsible for translating each of the WebDSL definitions resulting from the front end to corresponding Java classes.

The front end has weaved all aspect code in the definitions, simplifying the operations to be performed by the back end. For some WebDSL definitions the translation encompasses a one-to-one transformation: one WebDSL definition translates to one Java class. However, some elements of the source definitions are scattered throughout generated Java classes: a local-to-global transformation (see section 3.6). Similarly, for some WebDSL features many source definitions contribute to one Java class file (global-to-local). To ease the implementation of such one-to-many or many-to-one transformations, the compiler uses partial classes. Partial classes is a language feature that exists in recent versions of Visual Basic, C# and Smalltalk\(^1\), but not in Java (as of version 7). Multiple partial definitions of a class are composed together at compile time, very much like a simple form of aspects.

\(^1\)In Smalltalk, this feature is called Class Extensions.
Without partial classes during code generation, Java classes have to be transformed as a whole.

Typically, a class method in a generated Java class originates from a single AST node in the WebDSL AST. For example, the generation of a WebDSL entity property uses partial classes. The back end transformation rule has a `Property` construct as input, and results in a partial Java class, specified using concrete syntax:

```java
generate-entity-property(|x_entity_name|):
    Property(x_name, x_type) ->
    |[ @Partial
        class x_entity_name {
            protected x_type x_name;
        }
    ]|
```

After the back end transformations have been completed, the compiler performs a merging stage in which all of these partial classes are glued together to define complete Java classes. The result of the merging stage is a list of AST’s, each AST consisting of one merged Java class. A pretty-printer uses formatting presets to translate each AST into a text file. The pretty-printer adds spaces, indenting and line breaks according to the formatting settings, making the output not only machine-readable but also human-readable. While this step is not strictly necessary (since generated code does not always need to be human-readable), it does ease debugging of the generated application. As of the time writing, WebDSL does not provide a source level debugger, although one is in development [39].

### 6.3 Pipeline Separation

As figure 6.2 shows, the compiler pipeline can be subdivided in two sequences: the front end and the back end. This is a traditional division, in which the front end provides parsing, program validity checks and optionally transformations and optimizations. The back end receives a (simplified) AST for which it generates code in a target language. This separation stems from the fact that a compiler may have multiple back ends. For example, for GNU Compiler Collection (GCC) provides back ends for processor architectures like x86, Itanium and PowerPC [50].

While this two-phase architecture is very common, some compilers implement a three-phase architecture. The analysis phase (optimizations, transformations and such) is then extracted and decoupled from the front end. This allows multiple front end (and thus multiple source languages) to use the same analysis module and code-generating back end.

For WebDSL, the two-phase architecture suffices, as there is only one source language and the analysis is too language-specific to be shared with other compilers.

As with any pipeline, the output of the front end is the input of the back end. The output of the front end is a compiler-specific intermediate format. It consists of a collection of
top-level definitions in the WebDSL base language. For example, consider this input-output pair of the front end of WebDSL:

Input (WebDSL):

```markdown
Define ('index', Page(), Body(
    Table(...)
))
```

Output, aspects weaved in (base WebDSL):

```markdown
Define ('index', Page(), Body(
    AccessControlCheck(
        Call('checkUser', 'admin'),
    ),
    Table(...),
    LogAccess()
))
```

In this example, the source WebDSL definition (shown as AST) consists of a page declaration having a title “index” and containing a table. The output of the front end however includes an access control check; this check must have been specified as an aspect in another source definition and is weaved in during the desugaring phase. Also, some logging is applied to the page.

The intermediate format is just another AST, which can be written to disk. For a whole-program compiler, this provides an easy compiler optimization. When the intermediate representation of a definition does not change (compared to a previous compilation run), the back end does not need to consider that definition.

WebDSL compiler benchmarks have shown that a considerable amount of time is spent in the back end. Figure 6.3 shows the attribution of spent time to the two compiler phases. The figure contains benchmarks for projects of various sizes:

- **Hello World**, a simple example program distributed with the WebDSL compiler.
- **Reposearch**, a source code search application².
- **Yellowgrass**, an open source issue tracker³.

Generally, we’ve measured that about 70% - 80% of the time used by the compiler is spent in the back end.

---


³Project website is [https://github.com/webdsl/yellowgrass](https://github.com/webdsl/yellowgrass).
6. CASE STUDY: WebDSL

Because developers tend to compile often and therefore have minor changes between compilation runs, one can deduce that much of the work performed by the compiler is generating Java files that are already present and up-to-date from the last compiler run. Therefore, our first modification to the compiler was applying a caching mechanism on the back end of the compiler. The compiler writes all intermediate definitions to disk. After the front end completes, each input to the back end is compared to the last version (cached on disk); when the intermediate definition has not changed since its previous compilation, the back end simply ignores this definition. It will not generate Java files for these definitions. The Java files of the previous compilation run are still present and up-to-date. This basically follows the principles of memoization as described in section 3.2.1.

While this does not provide selective recompilation or incremental compilation, it does provide a major performance improvement for the compiler. Several problems however troubled this seemingly simple-to-implement caching solution.

![Figure 6.3](image.png)

Figure 6.3: Attribution of time spent by the WebDSL compiler in the front- and back end for projects of various sizes

6.3.1 Cached Generation

Conceptually, the usage of partial classes as described in section 6.2.6 means that there can be a many-to-one relationship between intermediate definitions and target files. For some transformations this indeed holds. For example, a Java class containing all page names (each page is a top-level definition in WebDSL) is generated using a many-to-one transformation. Mostly however, the set of partial classes for one class name all originate from the same intermediate definition. The generate-entity-property rule given above is one such
example: all partial classes are generated from elements within the entity’s intermediate definition.

We can therefore differentiate several classes of transformations:

1. One-to-one. One single intermediate definition is compiled to one Java class. Partial classes may be used, but all class parts are generated by the one intermediate definition.

2. One-to-many. One intermediate definition generates multiple Java classes, but none of this classes are influenced by other intermediate definitions.

3. Many-to-one. Using partial classes, multiple intermediate definitions contribute to one Java class.

With classes 1 and 2 we can assign one *owner* intermediate definition to each Java file, enabling the usage of caching in the back end as described earlier. For class 3 we need additional knowledge of which intermediate definitions contribute to what target files, a problem similar to the more general problem of separate compilation in the presence of aspect-oriented language features.

As separation the front- and back end was just a first step in optimizing the compiler, we decided to classify transformations based on the classes above and simply disable caching for the last class. Actually, the caching code enables caching on an opt-in base for the compiler developer; caching is enabled only on eligible transformations.

The interface of this caching scheme consists of only one rule: `cached-generate`. It operates on an intermediate definition and has two parameters: the back end rule that generates Java from the definition, and an unique cache name. The rule works only by providing the side-effect of writing Java files on disk; it does not modify the input definition. The signature is therefore represented as follows:

```latex
cached-generated(generate-rule | cache-name): 
   def -> def
```

The outline of the operation of this library rule is as follows:

1. Check cache validity.
   a) Read cache file with name *cache-name*. The contents of this file is a tuple, consisting of a compiler version identifier (*compiler-version*), a list of Java files (*list-of-java-files*) and the AST of the intermediate definition (*definition-ast*).
   b) Verify that *compiler-version* equals the current compiler version.
   c) Verify that all files in *list-of-java-files* still exist on disk.
   d) Verify that all files in *list-of-java-files* have a modification date equal to, or before the modification date of the cache file *cache-name*. 

e) Verify that the definition-ast, the AST in the cache file, equals the argument def of the rule invocation.

2. If all of the above checks succeed, the rule succeeds. The definition does not need to be recompiled to Java code, as doing so will result in Java files that are already present and up-to-date. The rule application of cached-generate succeeds without invoking the back end generation rule.

3. Otherwise, if any of the above checks fails, the generated Java files are considered out-of-date. Continue with the next step.

4. Execute generate-rule and thereby generate a set of (partial) classes.

5. Merge partial classes.

6. Write all generated Java files to disk.

7. Write cache file with name cache-name containing the current compiler version, the list of generated Java files and the definition AST.

For example, applying the caching scheme to the compilation of a page definition looks like:

```java
generate-page: Page(name, ...) ->
<cached-generate(page-to-java, name)>
```

The rule page-to-java is responsible for transforming an intermediate page definition to one or more Java files. The page name is used as the cache name; care must be taken that this name does not conflict with other top-level definitions, for example, with entity definitions. The generation and usage of the cache file is entirely abstracted away from the compiler source by using this library rule.

Figures 6.4 and 6.5 schematically show the usage of the cache file. The first figure shows the first compiler run, generating the cache file during Java generation. The second figure shows the usage of this cache file in a subsequent compiler run; the cache file is used to determine that the Java generation can be skipped.

### 6.3.2 Pruning

After the front end has generated a set of top-level WebDSL base language definitions, the modified back end now uses cache files to prevent the processing of intermediate definitions that have not been modified since the last compiler run, and would re-generate Java files that are already present.

For a particular intermediate definition, we can identify the following cases:
6.3. Pipeline Separation

1. There is no cache for this definition. The definition was not passed by the front end during the last compiler run. A change in the source program has resulted in the creation of a new definition. No caching can be performed, as there is no Java output from a previous compiler run.

2. The AST of the definition is equal to the cached version, stored by the previous compilation run. The compiler does not re-generate the Java code for this definition.

3. The AST of the definition does not equal the cached version; at least one change in the source program has caused this definition to be modified. New Java code has to be generated. The caching rules will apply the back end rule to generate appropriate Java classes, overwriting the existing Java files(s).

A last case occurs when the intermediate program misses a definition compared to the last compilation run. For example, removing a template from the source program results in the template not being present in the intermediate program, and the resulting Java file should be deleted. During compilation, the compiler maintains a list of Java files that either (1) have been generated this compiler run or (2) were not re-generated because of the caching algorithm. After the compiler has written all Java files, it scans the output directory for Java files. If a Java file name is not present in the compilers list of Java files, it is considered
6. CASE STUDY: WebDSL

“stale” and is removed. This ensures that Java files do not stick around when the user removes parts of his WebDSL program.

6.3.3 Deterministic Renaming

In section 6.2.2 we described the renaming step in the WebDSL compiler pipeline to give all variables, including local variables, a globally unique name with respect to the complete input program.

This approach enables the lookup of any variable without considering any scoping or shadowing rules of the language in any phase following the renaming phase. However, this renaming scheme imposes a problem with respect to the pipeline separation. Consider a source program consisting of 3 definitions $S_1, S_2, S_3$, each containing a local variable $x$. The renaming phase would rename this variables to $x_1, x_2, x_3$, in the order it encounters the variable declarations.

The names $x_1, x_2, \text{and } x_3$ will be used from the renaming phase onwards. After the front end has typechecked and desugared the input program, the resulting definitions still contain these variable names. Also, the cached AST’s of the definitions will contain these variable names.

Because of its occurrence in the intermediate AST, the numbering scheme influences the back end caching. Suppose the first definition $S_1$ in the source program (the definition containing the $x$ that was renamed to $x_1$) is removed from the source code. When the compiler renames the 2 remaining definitions $S_2, S_3$, the $x$ in $S_2$ will be renamed to $x_1$ and the $x$ in $S_3$ is renamed to $x_2$.

In other words, the numbering of the variables has shifted with respect to the last compilation run. For the back end caching algorithm, it now appears that both $S_2$ and $S_3$ have changed their AST, because the numbers of the variables are different. However, the variables are local variables and the numbers are just appended for the ease of the compiler’s implementation. For the resulting Java program, there is no functional difference whether the name of the variable is $x_1, x_2$ or any other name, for that matter.

For a given input definition, its resulting AST after renaming is not deterministic. Considering $S_2$, in which the variable $x$ was renamed to $x_2$, the particular renaming is chosen because another definition $S_1$ already resulted in $x_1$ being used. Because of such inter-definition dependencies, changing one source definition potentially changes a lot of intermediate back end definitions, depriving the efficiency of our back end caching algorithm.

For this reason, we introduced a new, more deterministic renaming scheme. In essence, we provide an unique name for each top-level definition, and prepend this name to the globally unique name of each variable. Table 6.1 shows a comparison of the non-deterministic and deterministic renaming schemes of the WebDSL compiler.

Because of the prepended definition name, removing or adding top-level definitions no longer influences the globally unique names of variables in other definitions.

---

4 The generation of this numbered variable names is performed in such a way that the uniqueness of each name is guaranteed; if there was a variable called $x_1$ in the source language, a different naming scheme would have been applied.
6.3. Pipeline Separation

Table 6.1: Comparison of the original non-deterministic WebDSL renaming scheme and the improved deterministic scheme.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Input name</th>
<th>Using old scheme</th>
<th>Using new scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template user</td>
<td>x</td>
<td>x1</td>
<td>templ_user_x1</td>
</tr>
<tr>
<td>Entity user</td>
<td>x</td>
<td>x2</td>
<td>ent_user_x1</td>
</tr>
<tr>
<td>Entity user</td>
<td>x</td>
<td>x3</td>
<td>ent_user_x3</td>
</tr>
</tbody>
</table>

This renaming scheme requires that each top-level definition has an unique name. For some language constructs, the WebDSL language semantics and constraints result in an easy choice of unique name. For example, all entities must have an unique name in the WebDSL language. A couple of considerations had to be made while determining a unique name for each top-level definition in the source language:

1. Constructs of different types can have overlapping names; for example, an entity and a template can have the same name. We prepend the names of such definitions with a prefix implying their type; ent_ for an entity, templ_ for a template and so forth.

2. Templates, pages and functions may be overloaded. Two templates with the same name might co-exist, be it that they have different formal argument types. We postfix the globally unique names of overloadables with their parameter types; for example, a template with name main and formal argument types int and string will be named templ_main_int_string.

6.3.4 Information Store

During the execution of the front end half of the compiler pipeline, the input program is parsed, typechecked and normalized to the core WebDSL language. The renaming phase changes the AST of the input program to use only globally unique variable names, as explained in the previous section. The collect phase of typechecking populates the information store of the compiler with various data about types and declarations. Constraint checking uses both the input program and the information store to find semantic errors in the input program, but does not change either the input program or the information store. Finally, normalization (desugaring) performs major refactorings on the input program AST, using the information store as a temporal place to store context information while performing transformation.

During this pipeline, the compiler maintains state in two distinguished formats:

1. For each top-level definition, the current version of the AST is kept in memory. This AST changes along the compilation pipeline. New definitions may be generated during transformation, or definitions may be removed from the input program.

2. The information store stores all data the compiler temporarily needs to stash away, to be used later in the pipeline. For example, typing information is kept in the information store and used by various stages of the compiler.
At the end of the front end pipeline, the compiler has a list of top-level definitions in the intermediate base language, and an information store containing facts about this definitions. Note that the entries in the information store are not bound to a top-level definition; they represent random pieces of information, some only relevant to a single definition, some globally relevant for the compilation of the whole program.

This introduces a problem for the separation of the front- and back end of the WebDSL compiler. The back end caching works by comparing the intermediate definitions that are input for the back end compiler. If such an intermediate definition has not been modified, the back end phase is skipped for that definition. But because the back end also reads information from the information store of the compiler, all information contained in the information store is actually an implicit dependency of the back end compilation. Figure 6.6 shows the implicit sharing of information between the front- to the back end.

The back end caching strategy assumes that the AST of an intermediate definition deterministically translates to Java code. When the back end uses information from the information store, this assumption no longer holds. Consider the following back end Java generation rule:

```java
generate-template:
  Define(x_name, args, body) ->
  | [ @Partial
    class x_name {
      protected Integer timestamp;
    }
  ]
  where <UsesTimestamp> x_name
```

![WebDSL compiler information flow](image)
The rule generates a timestamp variable in the Java code for a template, when the condition holds that a *UsesTimestamp* dynamic rule was set for the template name. This rule will have been set by the front end during typechecking or desugaring. Note that generation of the template Java class now depends on the state of the information store as filled by the front end.

For reliable caching, we need to truly separate the front- and back end. No information other than a definition’s AST can be implicitly shared between the two compiler parts, as this information sharing disables the usages of the AST-based caching mechanism.

We’ve separated the compiler in two parts at implementation (Stratego) level. This means that the compiler now consists of two separate executable programs, thereby disallowing any information to be shared except the data written to disk by the front end and read in by the back end. This makes it explicit what information is shared. This separation requires a major refactoring in the back end, as the context information store is now empty when the back end starts. Any information that is needed from the front end must now be included in the AST of the definitions that is passed from front- to back end. Because no information flows from the front- to back end except from the definition AST’s, we are assured of a deterministic and cacheable one-to-one mapping between intermediate definitions and Java output files.

The refactoring entails looking up all places in the back end where information from the front end is read from the information store. This context information is then explicitly added to the AST as a final stage of the front end. The *add-back end-annotations* rule migrates information from the context information store to the AST of a definition, where needed. Following the example of the timestamps above, the code below shows one particular implementation of that rule:

```stratego
add-back end-annotations:
  Define(x_name, args, body) ->
    Define(x_name, args, body){UseTimestamp}
  where <UseTimestamp> x_name
```

The rule adds an *UseTimestamp* annotations\(^5\) to the *Define* constructor, indicating that the template uses a timestamp. This fact is now represented in the AST and not only in the context information store. The code generating back end rule showed before can now be refactored to use this annotation:

```stratego
generate-template:
  Define(x_name, args, body){UseTimestamp} ->
  |
  @{Partial
  class x_name {
    protected Integer timestamp;
  
```

\(^5\)Using the ATerm storage format for AST nodes in Stratego, each node can have a number of nodes attached to it. These attachments are called annotations.
Figure 6.7 shows the differences in information flow caused by the compiler refactoring.

![Diagram of WebDSL compiler information flow](image)

Figure 6.7: WebDSL compiler information flow, old-new comparison

### 6.3.5 Performance

Using caching for the decoupled back end should speed up the compilation time significantly for incremental compilations in which not many input definitions have been modified.
Figure 6.8 shows a measurement of compilation time for a nearly empty “hello world” application. The left-most column “Without caching” shows how long initial compilation takes without caching enabled. The column is divided in two parts; front end and back end. The next column shows the same measurement for our improved compiler supporting back end caching. The caching, however, is only usable when there actually is a cache; i.e. only for subsequent compiler invocations. The difference between the left-most two bars indicates the overhead of separating the pipelines and keeping track of cache files.

The right-most column shows the performance measurements for a recompilation, using back end caching. One of the application’s input files has been “touched”: the modification timestamp of the file has been renewed, forcing the compiler to recompile this input file. The last column visualizes the effects of the back end caching. The time spent in the front end is approximately equal to the initial compilation run. The back end uses caching to determine that almost no output files have to be re-generated, because there are no actual changes to the source files. Therefore, the back end is highly efficient and the time spent in the back end can be neglected in comparison with the time spent in the front end.

Figure 6.9 displays the same benchmarks performed on the medium-sized WebDSL application Yellowgrass.

![Figure 6.8: Benchmarks of back end caching using a “hello-world” application](image)

### 6.4 Development Environment

The WebDSL compiler and accompanying toolset have traditionally been tools operated from the command-line. The `webdsl` command is a simple wrapper around Ant [21], an off-the-shelf build tool. Ant is mainly used for building Java projects, but can be instructed to perform arbitrary tasks. The tool expects tasks to be defined in a XML file, which also
6. CASE STUDY: WebDSL

Figure 6.9: Benchmarks of back end caching using a medium-sized application

specifies dependencies between tasks. Some tasks are built-in (like compiling Java classes and copying files) but more complex tasks can be provided by Ant plugins.

6.4.1 Command line usage

The workflow for developing a WebDSL application consists of these steps:

1. `webdsl new` creates a directory with some necessary files to start developing.

2. The application is implemented by creating WebDSL source files using the text editor preferred by the developer.

3. `webdsl build` invokes the WebDSL compiler and builds the application.

4. `webdsl test` runs all unit tests using an ephemeral data storage back end.

5. `webdsl deploy` takes the output of the WebDSL compiler and deploys the web application to a running web server.6

6.4.2 Spoofax

Aiding the developer with an easier workflow, the WebDSL team decided to build an integrated development environment (IDE) for WebDSL developers. Spoofax [29] is a language workbench that enables language developers to create IDE’s, without having to code the “boilerplate”. Basically, all IDE’s created with Spoofax share the same features, like

---

6 Currently, Apache Tomcat is the only supported web server.
syntax highlighting, semantic validation, code completion, reference resolving and code folding. Spoofax generates an IDE from a language description; a description of both the syntax and structure of a language, and its rules for typechecking. The output of Spoofax is not a complete IDE but rather an Eclipse plugin. By using Eclipse as base programming environment, Spoofax focuses on the demands for a specific programming language, rather than more commonplace matters like window management, file management and operating system compatibility.

Spoofax uses Stratego/XT to describe languages, the same language as WebDSL uses for its compiler. Spoofax uses SDF to specify the language’s syntax, as does WebDSL.

The syntax definition and grammar of a language are the most important input for Spoofax to generate an IDE. Syntactic validation can be performed by providing a SGLR grammar, for which Spoofax will automatically create an incremental parser, that validates files as the user types. By using heuristics, Spoofax will automatically generate rules for syntax highlighting and code folding. These rules can be customized by the workbench developer, if needed.

To provide semantic checks and reference resolving, the language developer can create language-specific Stratego/XT rewrite rules. For example, the check rule is invoked for each AST node and is used to collect semantic errors. This style of typechecking is also used for NWL (section 4.3.4) and WebDSL (section 6.2.3). Figure 6.10 repeats an example of a “check rule”.

```
check:
    FunctionCall(name, args*) ->
    "No function exists with this name"
    where not(<FunctionArguments> name)
```

Figure 6.10: Typecheck rule for a Spoofax-based IDE

### 6.4.3 WebDSL / Spoofax integration

Development of the Spoofax-based WebDSL IDE started in December 2011. At first, type-checking rules were added to the editors one-by-one, duplicating much of the compiler’s implementation of the typechecker. In preparation of supporting incremental typechecking in the IDE, we’ve refactored the editor to share much of its codebase with the compiler. Figure 6.11 shows how we reuse the compiler’s rewrite rule to enable typechecking “for free” in the editor implementation. The shaded boxes indicate non-custom Spoofax components.

When the compiler finds that any of the semantic consistency checks fails, a human-readable error message is printed along with the location (filename, line number) of the error. In the Spoofax IDE, this information is passed to the Spoofax framework. Subsequently, Spoofax translates the information about errors to the default Eclipse format. The Eclipse workbench finally gives the developer visual feedback about the failed consistency checks, as shown in figure 6.12.
6. **Case study: WebDSL**

![Diagram](image)

**Figure 6.11:** Sharing code between WebDSL compiler and Spoofax-based IDE

```
define showTask(t:Task)
header{
output(t.title)
}
output(t.description)
```

**Figure 6.12:** Eclipse IDE showing feedback for a failed consistency check
6.4. Development Environment

6.4.4 Towards an incremental typechecker

Quick feedback on semantic errors eases the job of developing an application [25]. While the Eclipse IDE is not language-bound, its was originally designed for Java developers. When developing Java application, a custom Eclipse Java compiler is used that is very fast and works incrementally (see section 3.4.2).

While a developer is typically typing in one file at a time, because of (transitive) imports and aspect weaving, the one file cannot be typechecked separately. The Spoofax workbench does not provide any support for incremental typechecking. We have incorporated the incremental typechecking algorithm in the WebDSL editor, based on the implementation provided by our work on an incremental typechecker for the experimental language NWL in chapter 4.

6.4.5 Editor pipeline

The editor pipeline is basically a shortened version of the WebDSL compiler pipeline. Spoofax does not have any features for analyzing multi-file projects, that is, languages that support importing. Because of that, handling imports and cross-file dependencies is implemented using custom code in the WebDSL editor.

1. Spoofax detects that a file has been modified; i.e. the user has typed in an open window showing a source file in the IDE.

2. The modified file is parsed by Spoofax using WebDSL’s syntax definition and grammar.

3. The rewrite rule editor-check is invoked with the AST of the modified file as input.

4. As described in chapter 3, the editor reads the automatically generated signatures of imported files. If a cached signature is not available, the imported file is parsed and the AST is stored along with the input AST.

5. Having one or more AST’s of files, the editor invokes the declaration rule of the WebDSL compiler. This is the first of two phases of typechecking: collecting information of types and structures in dynamic rules.

6. Phase two of typechecking entails using WebDSL’s consistency rules to find semantic errors in the input files.

7. The errors are collected and passed to Spoofax, which will mark them visually in the editor views.

The structure and contents of the cache files have been discussed in chapter 3. The Stratego/XT implementation of these steps follow the same implementation as we described in chapter 5 for the NWL language.
6.4.6 Editor particularities

There are a few remarks that apply specifically for the editor and not for the standalone WebDSL compiler. The standalone compiler performs a complete typecheck on each invocation, that is, all semantic errors are shown. Even in files that are not modified, semantic errors can arise because of changes in a different file.

For an editor, providing quick feedback is important, but only when it directly affects the user. The speed of feedback on the correctness of the file that is currently edited (and thus visible) to the user is far more critical than detecting errors in a file that is not currently opened. However, such errors do need to be displayed: they indicate the consequences of changes in the active file.

Using this observation, we’ve turned the incremental typechecking in a two-step typechecker. The first step uses cached signature to check the file in which a change has been made, in other words, the currently active file. Calculation of the impact to other files has less priority and is performed using a background task. The mechanism of enqueuing background tasks is a feature we’ve added to the Spoofax framework for this purpose. The background task first calculates a list of files that are impacted by the last typecheck run. Another background task is then started for the incremental typechecking of each of these files.

For the sake of responsiveness of the Eclipse IDE, we’ve chosen to perform typechecking of the active file as the user types. The enqueuing of background tasks to check impacted files is only performed when the user saves the file.

6.4.7 Signature Optimization

For the standalone WebDSL compiler, each entry in the information store that is added during the declaration phase of the typechecker is recorded by the dynamic rule tracking infrastructure. This allows a selective recompilation to use only the signature of a file, not the actual source file, in case a file has not changed and its compilation is not affected by current modifications.

For the editor, an optimization can be performed for storing the signatures. During a typecheck invocation on an active file, all signature files corresponding to that file’s (transitive) imports need to be loaded from disk. Therefore, keeping this signature files are small as possible can cause significant performance gains.

In the declaration phase of the typechecker, many dynamic rules are created. Some of these are used for the constraints phase of typechecking; some of them are only used later in the compiler pipeline for optimization analysis, or during model-to-model transformations. Some rules might be used both for constraint checking and further on in the compiler pipeline.

Particularly, dynamic rules that are not used during constraint checking are not of any relevance for the incremental typechecking of the editor. The remaining compiler pipeline phases following constraint checking are not performed by the editor, as its only concern is to retrieve a list of semantic errors found in the source file.

We have developed a tool that determines which of the dynamic rules are of importance
6.4. Development Environment

for the editor. These are the rules that are:

1. Created during the declaration phase of the typechecker

2. Used during the constraint checking phase of the typechecker

Note that the second predicate does not exclude rules that are also used further on in the compiler pipeline.

Our tool is written in Stratego/XT and performs static code analysis on the WebDSL compiler source. The tool therefore is a form of meta-programming; the software tool analyzes another software program. First, the tool combines all source files comprising the WebDSL compiler using pack-stratego, which reads all Stratego source files, processes imports and writes all rewrite rules to one packed Stratego file. This is a single-file representation of the functionality of the compiler. Using this file containing all rewrite rules, the tool statically determines the call graph corresponding to the declare rule, which is the starting point for the WebDSL typechecker declaration phase. For each rewrite rule in this call graphs, the set of dynamic rules added by that rewrite rule is determined. Then, a similar call graph is created for the constraint-error rule, the starting point for constraint checking. Because dynamic rules are rewrite rules as well, an application (usage) of a dynamic rule is translated to an entry in this call graph. The intersection of the created rules and the used rules forms the set of dynamic rules that need to be stored for incremental typechecking.

Summarized, these steps are performed by the tool to create a list of dynamic rules that are relevant for the signature files:

1. Create a packed Stratego file containing all rewrite rules of the WebDSL compiler.

2. Using this packed file, create a call graph starting with the declare rule. For example, declare applies declare-entity which in turn applies declare-entity-body, and so on.

3. For each rule in the call graph, determine which dynamic rules are created. For example, declare-entity-property creates TypeOf for each property of an entity, while declare-entity creates IsEntity indicating the presence of an entity type with the given name.

4. Create a call graph starting at constraint-error. This rewrite rule applies TypeOf to discover type errors in expressions and IsEntity to determine whether an identifier refers to an entity type. The set of rule names in this call graph comprise the set USE.

5. Create a set CREATE, containing all the dynamic rules created by rewrite rules in the call graph originating at declare. This set contains TypeOf and IsEntity.

6. Intersect the set CREATE with the set USE. These are exactly all dynamic rules created in the declaration phase of the typechecker and used by the constraint checking phase.
Figure 6.13 depicts the graph resulting from these steps. On the left, a small subset of the call graph originating from `declare` is drawn. On the right, the figure shows a part of the call graph for the constraint checking phase. On the bottom there is a shared dynamic rule, one which is relevant for the typechecker signature files.

The result of this tool is a list of dynamic rule names that are part of the typechecker signature of file. During incremental typechecking, each recorded rule is tested against this list of rule names. If a dynamic rule is not in the list, it is not recorded and therefore not stored in the signature file. Note that the list of dynamic rules is constructed from the compiler’s source; when the compiler is modified, the list needs to reconstructed by using the static analysis tool.

We’ve found that for a recent version of WebDSL, of the 165 dynamic rules created during the declaration phase, only 119 are used in the constraint checking phase. Therefore, this optimization decreases the size of signature files by about 28%, on average.

![Diagram of call graphs](image)

Figure 6.13: Subset of the call graphs used to determine which dynamic rules are relevant for the editor typechecker

### 6.4.8 Reference resolving

Reference resolving is a feature of Spoofax that allows the language developer to actualize references in the language. This feature is mostly used to visualize use-def relations (section 3.4). For example, an IDE can provide a shortcut that links the usage of a variable to the declaration of that variable. In Eclipse, holding the control key (or Command key, for Mac users) turns references into clickable links.

Spoofax provides the editor’s rewrite rules with an ATerm of AST of the edited file. This ATerm has special features. Invisible to the editor developer, each node of the AST
contains information about its origin: the source file and location from which the parser created the AST node.

This location information is used to implement reference resolving in an editor. For example, this declaration rule stores not only type information of a variable declaration, but also the declaration AST node:

```plaintext
declare: v@| [ var _name : _type; ] | -> v
with rules {
    TypeOf : _name -> _type
    DeclarationOf : _name -> v
}
```

While the typechecker uses the `TypeOf` rule, the rewrite rule for reference resolving makes use of the `DeclarationOf` dynamic rule to lookup the declaration for a variable reference. Whenever the IDE signals a click on a variable usage, Spoofax invokes the `editor-resolve` rewrite rule of the WebDSL editor. This rewrite rule can use the `DeclarationOf` dynamic rule in this manner:

```plaintext
editor-resolve: VariableAccess(_name) -> declaration
with declaration := <DeclarationOf> _name
```

Spoofax receives the AST node containing the variable declaration. Using the node’s origin information, it is able to instruct Eclipse to show this declaration in the user’s viewport.

### 6.4.9 Inter-file reference resolving

This method of providing reference resolving rules to Spoofax is designed to accommodate intra-file reference resolving: where the target of reference resolving is contained in the same file as the clicked reference.

The reason why this does not work for inter-file resolving lays in the internals of Spoofax parsing. There are two types of parsing:

- Parsing invoked by Spoofax: when a file needs to be analyzed, either for typechecking, reference resolving or any other need, Spoofax automatically parses the source file and provides the editor’s custom rewrite rules with the AST. Each node of this AST has origin information.

- Parsing invoked by the rewrite rules themselves. When typechecking a file, the typechecker needs signatures of imported files. These can be provided by signature caching, or by parsing the referenced import file, when a cached signature file is not available.
In the first typechecking run, there are no cached files available. These are automatically created by our selective recompilation library discussed in section 5. For this first run, all imported files need to be parsed as their AST is not yet available. Spoofax only provides the AST for the “current” file, in which the change has been made that included the need for typechecking.

The editor’s rewrite rules, in this case contained in our library, parses other files using the Spoofax parser. As a result of that, the parsed AST contains origin information on each term. However, when writing the signatures to disk, the origin information is lost. This stems from the fact that Spoofax’s output format (binary ATerms) does not provide any means to write origin information to disk.

We’ve implemented a workaround for this shortcoming using a combination of our automatic signature tracking algorithm and the dynamic rule tracing implementation. This works by taking advantage of the fact that dynamic rules created during a typecheck invocation of Spoofax are retained until and during the following reference resolving actions. Let us return to the example above with the DeclarationOf rule creation during typechecking.

When this dynamic rule is created during the declaration collection phase of typechecking, the separation library code dealing with automatic signatures traces this write to the information store. The library keeps track of which source file contained the declaration from which the dynamic rule originated. For a declaration of globalVar in a module main.app, such a mapping is stored as follows:

\[(\text{DeclarationOf, "globalVar"}) \rightarrow \text{"main.app"}\]

Now suppose the reference resolver is invoked on variable reference of globalVar which is contained in a different source file: mod.app. During the invocation of the editor-resolve rewrite rule described above, the rule DeclarationOf will be used to find the declaration site of the variable. The usage of this dynamic rule is traced by the rule tracing implementation of the dynamic rule store. Using this rule tracing information combined with the dynamic-rule to source-file mapping, the reference resolver comes to know that the variable declaration is in fact located in main.app. This file is now parsed and searched for an AST node containing the declaration. By returning this node to Spoofax, the IDE knows which file to open at which line number. Summarizing:

- Spoofax invokes typechecking on the file that the user just modified.
- The WebDSL typechecker reads all imported source files that are needed for global consistency checks.
- The separation library traces rule creation during the declaration phase of typechecking, creating a “dynamic-rule to source-file” mapping.
- When resolving a reference, a dynamic rule is used to find the declaration site of the reference.
6.5 Library Integration

- Using the dynamic-rule to source-file mapping, the corresponding source file is determined.

- Parsing this source file leads to an AST with origin information, which can be passed to Spoofax.

This techniques allows the WebDSL editor to provide inter-file reference resolving without altering either the resolving rules or the Spoofax framework. Like the separation library, there is no WebDSL-specific code in the Stratego code that allows inter-file reference resolving. Therefore, any Spoofax-based editor should be eligible to use this implementation as-is.

6.5 Library Integration

Working on the WebDSL editor, we’ve used the separation library to implement incremental parsing and typechecking with automatic dependency analysis. The next step of the case study is to incorporate the full separation library in the WebDSL compiler. That is, not only to provide selective recompilation in the typechecker phase, but also in the model-to-model transformation phase of the compiler. Due to the pipeline separation we’ve discussed in section 6.3, the back end of the compiler is decoupled and the back end is only invoked for top-level definitions that are outputted by the front end. Therefore, the front end, including model-to-model transformation, performs the selection of which parts of the program have to be recompiled.

Integration of the selective recompilation library into an existing compiler was straightforward for our NWL+Aspects sample compiler. The compiler implementation followed the outline given by the generic compilation model described in section 3.1. As we have described earlier in this chapter, the WebDSL compiler follows the same principles as the NWL compiler for parsing, import management, typechecking, model-to-model transformations and code generation. Even though, there are some important differences. Most of these differences stem from the fact that the WebDSL compiler is actively developed by a number of people and actively used by web application developers. This has led to a fast release schedule, resulting in many ad-hoc solutions in terms of compiler design.

6.5.1 Pipeline

The basic pipeline of the WebDSL compiler was depicted in figure 6.2. However, the actual compiler implementation has some “quirks”. First, as section 6.2.5 introduces, access control weaving is performed in a separate phase. This introduces an implicit ordering in the compiler, namely that all other model-to-model transformations have been performed before access control is weaved in. This assumption is enforced by having two phases, which is encoded in the compiler’s implementation. Such an assumption should be compelled by using the context information store, allowing the selective recompilation library to trace this dependency. Next, the pipeline is not entirely one-way, in terms of ordering. During many model-to-model transformations, renaming is performed on newly generated
WebDSL elements. Also, declarations can be emitted from the model-to-model phase, effectively jumping back to the declaration phase, as the next section describes.

6.5.2 Emitting

During transformations in the front end, a source definition can emit another definition. For example, a declaration that enables access control in a WebDSL application emits a default login and logout page. The mapping between source definitions and intermediate definitions is therefore injective, but not surjective; it is 1-to-n.

During the model-to-model phase, (partially) transformed source definitions and generated intermediate definitions are present side-by-side. There are however differences in how the selective recompilation library must handle the generated definitions. We define the owner relationship as follows:

**Definition 13.** A source definition $S_i$ is the owner of a intermediate definition $I_i$ when:

- $I_i$ is the output of front end transformations on $S_i$, or
- $I_i$ is emitted during the transformations of $S_i$.

Each $I_i$ has only one such owner.

Using this definition of an owner, some addition logic has to be added to the selective recompilation library, for example:

- Entries in the context information store resulting from transformations on a generated intermediate definition must be treated as if they were the result of a transformation on its owner.
- Detect when an emitted definition is generated during a previous compilation run, but not during the current (pruning).
- For an emitted definition of which the owner is not present in source form, the compiled target file must be deleted.

The inclusion of emits in the generic compiler model requires more complex algorithms and bookkeeping for selective recompilation, needing a sound analysis of the problems arising from emitting.

6.5.3 Module Management

The present WebDSL compiler handles its source program as one huge AST: a top-level application node with each module as a child node (see figure 5.2). Our compiler model for an selective compiler actuates on a set of definitions: each module is broken up in top-level definitions, which are the unit of operation. During compilation, the selective recompilation library decides whether module definitions are to be added to the set of to-be-compiled definitions.
6.5. Library Integration

We’ve had to refactor each step in the compiler pipeline to operate on definitions, instead of operating on an entire application. Mostly, this means replacing one operation by an iteration over all definitions currently selected for recompilation. However, some operations require access to the complete AST, for example, checking whether two functions in the source program have the same name. This is implemented by traversing the entire AST, marking each occurrence of a global function name and checking for duplicates. The refactoring of the compiler included changing this way of finding duplicates. In our version, we use the declaration phase to record all global names, and find duplicates using these values collected in the context information store.

6.5.4 Current State

As discussed, there are some huge and some minor differences between the compiler model prescribed by the selective recompilation library and the model used by the current WebDSL compiler. Some of these differences have been resolved by refactoring the WebDSL compiler. Some others, however, like the aggregation of the model-to-model and access control phases, are beyond the scope of our research. Finally, emits are not accounted for in our algorithms, and are still an open issue.

A proof-of-concept incremental WebDSL compiler has been developed as a branch of the production compiler. While the incremental version has been refactored in its majority, a full compilation run (without cached information) produces the same output as the production compiler. To test the new compiler version using its new selective recompilation features, some example projects have been created and ran through the test suite described in section 5.4. Basic incremental functionality has been proved to work; unchanged source files will not be recompiled, while aspect weaving still works, basic dependencies are tracked and compilation is triggered when (transitive) dependencies require it.

However, the incompatibilities of the WebDSL compiler and our framework cause trouble for programs using the more complex constructs of the WebDSL language. For larger programs, the issues described in section 5.3.3 start boiling up; the storage format of the dependency information is not efficient enough. The amount of lookups for the WebDSL compiler is much larger than for the NWL compiler, which becomes an issue when the program size grows to ten of thousands of lines of code.

Summarizing, the incremental WebDSL compiler is not ready for production usage as of now, both due to the design of the WebDSL compiler and due to some unresolved issues with our approach. Our refactored WebDSL compiler, however, exemplifies how a compiler can be (partially) tailored to match the generic compiler model, when it does not natively.
Chapter 7

Related Work

Much work has been published on aspect-oriented languages, their features, implementations, benefits and their issues. On the contrary, the body of published work dedicated to solving the problem of incremental compilation for aspect-oriented programming languages is not substantial. There are however several authors proposing ideas that indirectly relate to incremental compilation.

7.1 Dynamic weaving approaches

Many aspect-oriented languages provide load-time or dynamic weaving, as described in chapter 2. At first sight, choosing a dynamic weaving approach seems to alleviate problems related to compilation times. While AspectJ was originally implemented using static weaving, newer versions also provide a dynamic weaving approach [4]. For program execution, dynamic weaving is indeed an easy way of solving the incremental compilation problem: their simply is no compilation involved with aspect weaving. This is similar to an interpreter, which has no problems loading and executing huge programs, as it does not need to analyze the program as a whole; it only needs to load program constructs on-demand, when they are executed. Besides runtime performance, there is a fundamental problem that is shared between interpreters and aspect-oriented languages performing dynamic weaving. This problem is that such runtime-centered environments are not able to provide static information about a program. This problem manifests itself in two ways, at least:

- During development of a program, there is little to no static consistency feedback. For example, type errors cannot be found by an interpreter until the program is actually executed and reaches the point of failure.

- As a result of that, interactive development environments (IDEs) cannot provide interactive feedback of semantic consistency. Furthermore, more advanced editor features like incremental typechecking, reference resolving, code outlining and code browsing cannot be provided if there is no static program analysis.

In practice, developers often require these advanced IDE features to be available in order for a programming language to be a success. Often, an interpreted language therefore
is (possibly afterwards) supplemented by a compiler, to be used for providing static analysis from the IDE. The front end of the compiler suffices: code generation is not needed as the interpreter can be used to execute programs.

Going back to aspect-oriented programming, the case is very similar. To provide a feature-rich IDE, static analysis is required on the source program. The AspectJ Development Tools [12] are an example of an IDE providing aspect-related feedback to the developer. Also, existing Java-related editor features are modified to “recognize” aspect-oriented features. For example, the release notes of version 2.1.1 [51] describe the ability of existing Java tools like identifier search, navigation and hovering to co-operate with inter-type declarations.

Language developers can use dynamic weaving during program execution, but static analysis will still be required to provide a fully-fledged IDE. Therefore, dynamic weaving does not solve our initial problem of providing an incremental compiler for aspect-oriented languages.

7.1.1 Using an Aspect Interface

The work of Rajan, Dyerm Hanna and Narayanappa [48] describes a runtime weaving approach to “preserve separation of concerns through compilation”. The authors argue that while separation of concerns is traditionally a source-level concept, the separation should be preserved even after compilation. By using aspects the concerns are separated at the source level and preserved during the analysis, design and implementation phases of developing software. However, the separation ceases to exist during the compilation phase and at runtime. This is inherent to the technique of weaving aspects at compile time. Different issues with this approach exist and warrant the investigation of alternative, particularly the difficulty to provide an incremental compiler.

An interesting side node is that while aspect-oriented programming is a relatively new technique, similar problems also exist for traditional programming techniques: they tend to arise whenever the source language of a compiler has features not available in the target language. One of the examples given is a procedural language which compiles to a monolithic instruction set. This is accomplished by in-lining, scattering method bodies throughout the system. This very much resembles the weaving process, in which aspects are scattered throughout the target code.

The approach taken by the authors of is to define an interface between the compiler and the target environment, which must be an interpreter. Using the .NET framework bytecode as target language, the interface consists of two new bytecode instructions: bind and remove. The former takes a string which specifies the pointcut (the join points at which advice should be applied). It instructs the runtime environment to make all changes necessary to execute the advice code when a join point is hit. The latter instruction takes an aspect (actually, the advice’s “delegate”) and removes the binding with the pointcut.

Adding these two instructions the target language relieves the compiler from weaving altogether, alleviating the loss of separation in the target code. If the source code has to be compiled into a language readable by the interpreter, the new target instructions make sure that advice code is not woven into and scattered across other target modules. This way,
adjusting a pointcut, which leads to full compilation in most (if not all) aspect-oriented language compilers, results merely in a mutated string in the target bytecode.

The new approach for aspect-oriented compilation is based on delayed runtime weaving. This is accomplished by a very simple interface of just two instructions between the compiler and the runtime environment (the interpreter). To make use of the approach, besides defining the interface, one would also be required to change both the compiler and the runtime environment to use this interface. Modifying the compiler is relatively simple, because adding aspects consists of inserting bind calls in the initialization of a system. Currently, the popular aspect-oriented language AspectJ does not support runtime advice-removing, so remove would be unused for such an implementation. However, changing the runtime system is far more challenging. The Steamloom project [7] extends the Jikes Research Virtual Machine by adding runtime aspect weaving. This is accomplished by modifying some internal data structures to point to advice-code to be weaved in.

While this approach is independent of the source language, it is dependent on both the compiler implementation (which needs to add extra bind instructions) and the runtime environment, where two new bytecode instructions have to be implemented. Furthermore, the approach inherits the general problems we’ve addressed concerning dynamic weaving.

7.1.2 AspectCOOL

AspectCOOL is a general-purpose aspect-oriented language providing functionality for weaving aspects in programs written in the general-purpose (research) language COOL. The implementation of COOL was mainly an experiment to evaluate development using a language design tool called LISA [41].

Besides the basic aspect-oriented features that most aspect-oriented languages provide, AspectCOOL provides separate compilation of aspects and components, which improves reusability: for example, aspects can be weaved in commercial off-the-shelf applications. Selective recompilation required the base language COOL to be extended with dynamic (on-demand) module loading, method call interception, reflection (based on metaobject protocols [32]) and “slots”.

Slots represent explicit code points at which advice (the semantics of an aspect) can weave in. A slot appears at statement level and consists of a name and a number of values: variables that are in (static) scope at the point of slot definition. An aspect can define a method which is executed with the given values for the parameters when execution reaches the point of slot definition. If no aspect defines behavior for a slot, it effectively turns into a no-operation.

The language strictly separates aspects in two parts: aspect classes and aspect applications. Aspect classes define the behavior of aspects, like advice in AspectJ. The aspect application part of an aspect defines the pointcut at which it must be weaved in the program. The reason for this separation is to prevent (compile-time) dependencies between the two: if an aspect’s behavior changes only the aspect code has to be recompiled, thereby avoiding the reapplication of all aspect weaving. This idea is very similar to the work of Beugnard [6]. While this separates aspect code from pointcuts, it does not relate to the separate compilation of aspects and base code, which still need to be weaved at some point.
7. RELATED WORK

The application part of aspects contain the pointcuts that define a mapping between join points in target code and advice. Pointcuts are defined by wildcards, similar to AspectJ’s implementation, but use method transition (defined as a pair of methods) as the fundamental building block for defining join points. For slot advices, matching an advice to a slot is easy, as both can be unambiguously identified by their qualified name: slots are unique within both the class and aspect namespace. Additional language constructs indicate how many instances of aspect classes are to be instantiated.

The binding of advice to join points is done at runtime. This is possible because the COOL language provides method call interception as a language feature. This means that AspectCOOL performs dynamic weaving, which makes separate recompilation much easier to implement.

7.2 Observers and Assistants

Our research question has revolved about the feasibility of a technique to allow selective recompilation for a statically weaved aspect-oriented language, independent of the language under compilation and the compiler’s implementation language. This has led to very generic solutions focusing more on the algorithmic side of the problem. We’ve seen in chapter 6 that an actual implementation for a huge and complicated compiler does not come cheap. The work presented here provides (partial) solutions to the selective recompilation problem, but are more restrictive in the applicability of the solution.

Clifton and Leavens propose several changes to the AspectJ language to enable modular reasoning for systems using AspectJ [14]. First, it is established that Java programs (without aspect-oriented features) support modular reasoning\(^\text{1}\). By looking at a method, we need to look only at the method itself, the methods it calls and the superclasses of the object to reason about its functional behavior. Because of the explicit import and extends keywords in Java, we can statically construct the behavior of the method without considering all modules. This reasoning can be formalized by using JML, the Java Modeling Language. In the absence of aspects, Clifton and Leavens conclude that modular reasoning is possible because of the explicit references between the Java source files.

Aspects in AspectJ however are weaved in the modules at compile-time [27]. The modules and methods to which advice is applied are determined by matching the pointcut of an aspect to the available join points. Opposed to the normal Java source, a module does not have any reference to the advice that is to be applied to that module. By looking just at one module, it is not possible to reason which aspects are applied to it. Whole-program analysis is needed which considers the target modules, the modules it depends on and all aspects. Modular reasoning is therefore impossible with AspectJ systems.

A proposal is made to categorize aspects into two sorts. Observers are aspects which do not modify the state of the module on which they act. Examples include logging, statistics gathering, tracing and other simple aspects. Because observers do not modify the state and behavior of a module, their presence does not have to be considered in semantic analysis.

\(^1\)Section 2.2 describes the concept of modular reasoning.
The proposal annotates these aspects with a special keyword so that the absence of state mutation can be used as knowledge in analysis.

On the other hand, many aspects do change the state or behavior of the module in which they are weaved. These are classified as assistants. Modular reasoning requires the knowledge of which assistants are concerned for a given module. By introducing the accepts keyword, a module can explicitly declare from which aspects its accepts assistance. Hereby a static link is established between the module under reasoning and the aspect module containing behavioral changes. By carefully composing the specifications of the client module, the implementation module and the aspect-modules, the applicable complete specification can be obtained without whole-program analysis\(^2\).

The issue of modular reasoning is important in more than one field of research. First of all, by which the paper motivates its proposal, the decomposition of a system in modules should allow for easier understanding of parts of the system, because of the smaller size and independence. But not only humans suffer from the lack of modular reasoning in aspect-oriented systems. Computers generally have enough memory to analyze systems as a whole. Even though, separate analysis of modules is important because whole-program analysis and whole-program compilation are complex tasks and commonly use a lot of resources. Developers nowadays require instant feedback when developing a module, consisting at least of syntax and type-check errors. With non-modular reasoning, the type-checker does not know the dependencies between modules, resulting in a forced whole-program analysis.

The approach taken by this paper has several implications and limitations. There is no solution to the problem of automatic classification of aspects, so the developer is nominated to identify the aspects as being either an observer or an assistant. Requiring the developer to distinguish between observers and assistants is both unwieldy and error-prone. Each change to the code of an aspect possibly promotes that aspect from observer to assistant, or the other way around. Inadvertently classifying an aspect as observer invalidates the specification composition; modular reasoning is then likely to yield wrong results. One solution would be the automatic classification of aspects, or similarly the verification of “observerness”. This allows for static checking of aspect constraints, but depends on side-effect and alias detection. The paper only briefly mentions this possibility before disposing the issue altogether by referring to “a substantial body of work” that has been done on alias control.

The need for explicit declaration of accepted advice in each module puts an additional burden on the developer. In comparison to the pattern-matching approach taken by AspectJ to match a pointcut to the join points, there are both advantages and disadvantages. Without explicit acceptance from a module, it is easy to specify large amounts of classes for aspect weaving. One useful example includes the logging aspect, which might be applied (in the extreme case) to all methods. However, the paper’s approach does not impose any difficulties here because the logging aspect is classified as an observer, which does not need explicit acceptance. The question that remains is whether we want to support implicitly applying assistance aspects on modules. For example, consider a debugging aspect. It could consist

---

\(^2\)This composition of specifications is formalized by defining the creation of a specification composition graph.
of a before advice (executed before method invocation) that throws an exception to halt the system, whenever some breakpoint condition is true. This aspect clearly is assistance (it modifies the behavior of its target component) and therefore, in order to have modular reasoning, needs to be accepted by every module. Adding an accepts definition to every source file is required to enable this aspect, while in standard AspectJ a simple pointcut matching all methods would suffice. On the other hand, the string matches of AspectJ enforce a specific naming of Java classes. This could result in unintuitive class naming. Also, when adding new classes or methods to a system with existing aspects, it is important to keep in mind that newly created methods might match unrelated join points; the developer should be aware of all pointcuts and the string matches they contain. Using appropriate tools however, the developer can be alerted of this effect: for example, some editors have been extended to show which aspects match where. This particular problem is alleviated by the explicit acceptance of advice.

Filman and Friedman propose a number of language characteristics to distinguish aspect-oriented languages [20]. Two major language properties are quantification and obliviousness, which are necessary for aspect-oriented programming according to the authors. The concept of obliviousness revolves around the knowledge in advance on aspect application [19]: the places in code which “receive advice” should not have to be specifically prepared for this reception. The statement that “better AOP systems are more oblivious” seems to be in direct conflict with the proposal of observers and assistants: introduction of the accepts keyword destroys all obliviousness of the aspect-oriented language.

Classifying observers and assistants to provide modular reasoning may improve readability of source code and modular reasoning, but is also puts a considerable burden on the developer: identifying which aspect is eligible to be an observer and which are not. Furthermore, obliviousness, one of the basic concepts of aspect-oriented programming according to some, does not allow the explicit acceptance of aspects, as is required for assistants.

A further restriction on the usage of aspects is proposed in a case study by Kersten and Murphy [31]. Their work builds on the idea of observers and assistants. The behavior-modifying aspects (assistants) are restricted to be used with “after-constructor” join points. The advice is then executed just after object creation by the constructor of the object type. These aspects are called factories; they assist the construction of a module by extending or modifying it.
Chapter 8

Conclusions and Future Work

This chapter gives an overview of this project’s contributions. After this overview, we will reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

8.1 Contributions

We’ve analyzed the domain of incremental compilation in the presence of complex interdependencies, more specific, in the presence of static aspect weaving. While many developers in the aspect-oriented field have resorted to runtime weaving, thereby side-stepping the problem altogether, sooner or later developers using a language will request an interactive IDE. This IDE has to perform static analysis on the input program, revitalizing the need for static weaving.

Of the languages that do use static weaving, very few support incremental or selective recompilation. Even for the AspectJ language, having received much academic attention and detailed descriptions of its inner workings, there is no publication on incremental compilation. We provide a detailed research on language-agnostic dependency tracking, both for two-pass compiler phases as for exhaustive traversals of the source AST. The algorithms resulting from this research form the basis of our software framework.

We propose a framework that automatically keeps track of dependencies and instructs the compiler to incrementally select which parts of the source program have to be recompiled. The implementation of this framework in Stratego/XT has been integrated in a sample language (NWL) and in the WebDSL editor and compiler, showing the viability of such a framework.

By using a test framework, we can easily simulate complex cases in which dependencies “ooze” through many modules due to the transitive property of dependencies.

For a detailed overview of the software code contributions related to this research, see chapter 9.
8. Conclusions and Future Work

8.2 Conclusions

Our framework has been successfully implemented in the NWL pedagogic language and the WebDSL editor. The former proves the theoretical attainability of a language-agnostic framework; the latter proves the practical use for a usage and complex language.

Integration the framework in the standalone WebDSL compiler revealed some major issues, both on the WebDSL compiler side and on the framework side. Our algorithms use a generic compiler model, which can be mapped roughly on most compilers. We’ve seen, however, that the implementation of the WebDSL compiler does not match exactly enough. For these mismatches, two solutions present themselves: refactor the WebDSL compiler to match the generic compiler model, or change the generic compiler model. The first solution admits that the general compilation model is not general enough; apparently, it imposes more restrictions on the family of compilers it supports then we expected. This conflicts our initial goal of providing selective recompilation as a separate concern, because the language developer needs to take care in following our compilation model. The second solution, however, complexes the model of compilation and therefore our algorithms. Care must be taken not to add support of implementational details of a specific compiler to our library, which is again in contradiction to the Independence of our algorithm and the compiler.

The Stratego/XT implementation is functional, but has a serious shortcoming: it does not scale well when the input program becomes large. The implementation lacks a scalable storage back end for the dependency information. This information is now read in memory as a whole, letting the compiler scale linearly in memory usage with respect to the input program size. An incremental compiler should induce memory usage linear to the size of the changed parts of the input program. While this is a major practical issue, it does not affect the dependency tracking algorithms. The library code is easily refactored to use a different back end, when an efficient random-access storage back end for Stratego/XT arises.

8.3 Future work

The previous section can be summarized in two important questions:

- How to deal with discrepancies between the compiler model used by our library and actual compiler implementations?
- How to make the library implementation more efficient in terms of storage and memory usage?

The first question is an open question of which the answer determines whether the compiler developer is burdened with refactoring the compiler, or whether our library has to be extended to support more compiler designs.

The second question calls for additional research to the exact requirements of the storage mechanism for the dependency information. Possibly, different bookkeeping tasks of our library have different requirements, which will lead to multiple storage back ends. While
more efficient storage is a must-have for the compiler to lower the memory load, more
directions can be explored. For example, the library currently tracks all context information
store accesses. When using Stratego/XT as the implementation language for a compiler,
this translates to all accesses to dynamic rules. However, not all dynamic rules are actually
used outside of a single definition. Information that “remains within the definition” does not
introduce any dependencies and therefore can be ignored by the library, resulting in smaller
files for dependency tracking. A very basic form of such an optimization has been described
for the WebDSL editor, however, many more optimization are possible. Additionally, the
implementation of a compiler could mark entries in the information store as either private
or public, the former indicating information local to the current definition. Only public
information store accesses have to be tracked.

On the whole, we’ve seen that the two-phase typechecking compiler model is elegant,
simple to implement and easy to generalize. Besides, all Spoofax-based IDE’s already have
such a design. The separation library is easy to incorporate in this phase, exemplified by the
usage in the WebDSL editor. However, the model-to-model phase throws up a number of
problems with our approach.

A hybrid solution would be to use static weaving for editor purposes, but skip the model-
to-model phase in the IDE; for most tasks like reference resolving, auto-completion and
consistency checking, merely the typechecking phase produces sufficient information. Dur-
ing program execution, the runtime environment would perform dynamic weaving to side-
step the incremental compilation problem. Such an approach combines the advantages of
both static and dynamic weaving, choosing which technique to use on a per use-case basis.
List of Software Contributions

This chapter lists the software code contributions provided by this research and the online locations of this software. All software may be used freely for any purpose.

- Separate compilation library for Stratego/XT. Described in chapter 5.  
  https://svn.strategoxt.org/repos/WebDSL/webdsls/branch/lib-sepcomp

- NWL compiler implementation, integrated with our library. Described in chapter 4.  
  https://svn.strategoxt.org/repos/WebDSL/webdsls/branch/nwl-aspects

- WebDSL compiler improvements:
  - Pipeline separation (section 6.3)
  - Bugfixes
  - Decoupling of code for re-use in WebDSL editor
  - Addition of date picker
  https://svn.strategoxt.org/repos/WebDSL/webdsls/trunk

- WebDSL editor. Described in section 6.4. Implementation of:
  - Incremental typechecking, using our library
  - Multi-file typechecking
  - Inter-file reference resolving
  https://svn.strategoxt.org/repos/WebDSL/webdsl-editor/trunk/webdsl.editor

- WebDSL selective compiler (chapter 6). A start for integrating the library in the WebDSL compiler.  
  https://svn.strategoxt.org/repos/WebDSL/webdsls/branch/separate-compilation
9. List of Software Contributions

- Stratego/XT - Java back end. Addition of support for streaming ATerm format, fixing support for binary ATerm files and other bugfixes.
  https://github.com/metaborg/strategoxt/tree/master/strategoxt-java-backend

- Stratego/XT - C back end. Support for rule overriding, as described in section 5.2.1.
  https://svn.strategoxt.org/repos/StrategoXT/strategoxt/branches/strategy-overriding
Bibliography


